编程狂人

Programming Madman

NO.50

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/546a9adad91b14682f01d88c

欢迎下载推酷客户端体验更多阅读乐趣

目录

再谈Javascript原型继承

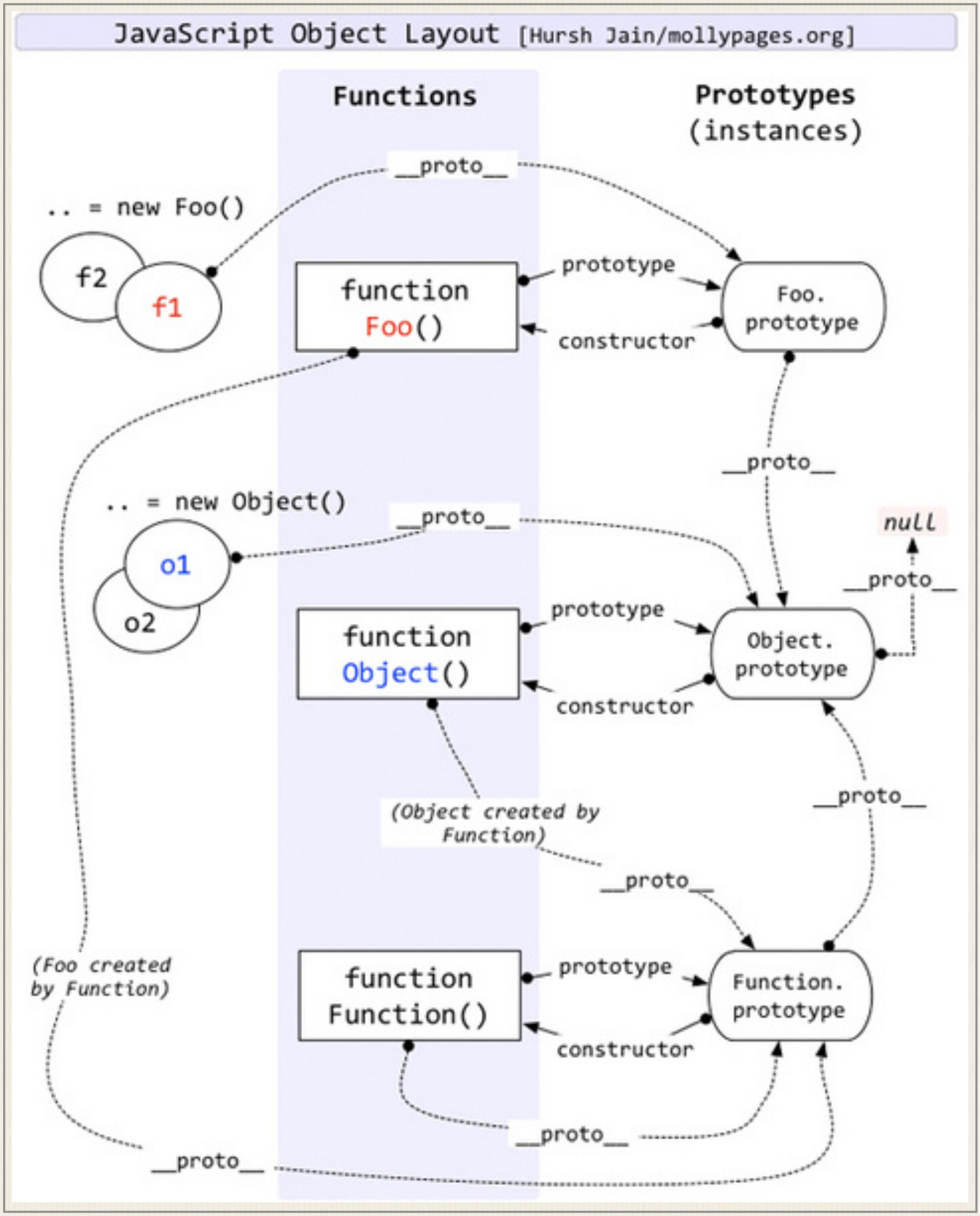
作者：lakb248

真正意义上来说Javascript并不是一门面向对象的语言，没有提供传统的继承方式，但是它提供了一种原型继承的方式，利用自身提供的原型属性来实现继承。Javascript原型继承是一个被说烂掉了的话题，但是自己对于这个问题一直没有彻底理解，今天花了点时间又看了一遍 《Javascript模式》中关于原型实现继承的几种方法，下面来一一说明下，在最后我根据自己的理解提出了一个关于继承比较完整的实现，如果大家有不 同意见，欢迎建议。

原型与原型链

说原型继承之前还是要先说说原型和原型链，毕竟这是实现原型继承的基础。

在Javascript中，每个函数都有一个原型属性prototype指向自身的原型，而由这个函数创建的对象也有一个__proto__属性指向这个原型，而函数的原型是一个对象，所以这个对象也会有一个__proto__指向自己的原型，这样逐层深入直到Object对象的原型，这样就形成了原型链。下面这张图很好的解释了Javascript中的原型和原型链的关系。



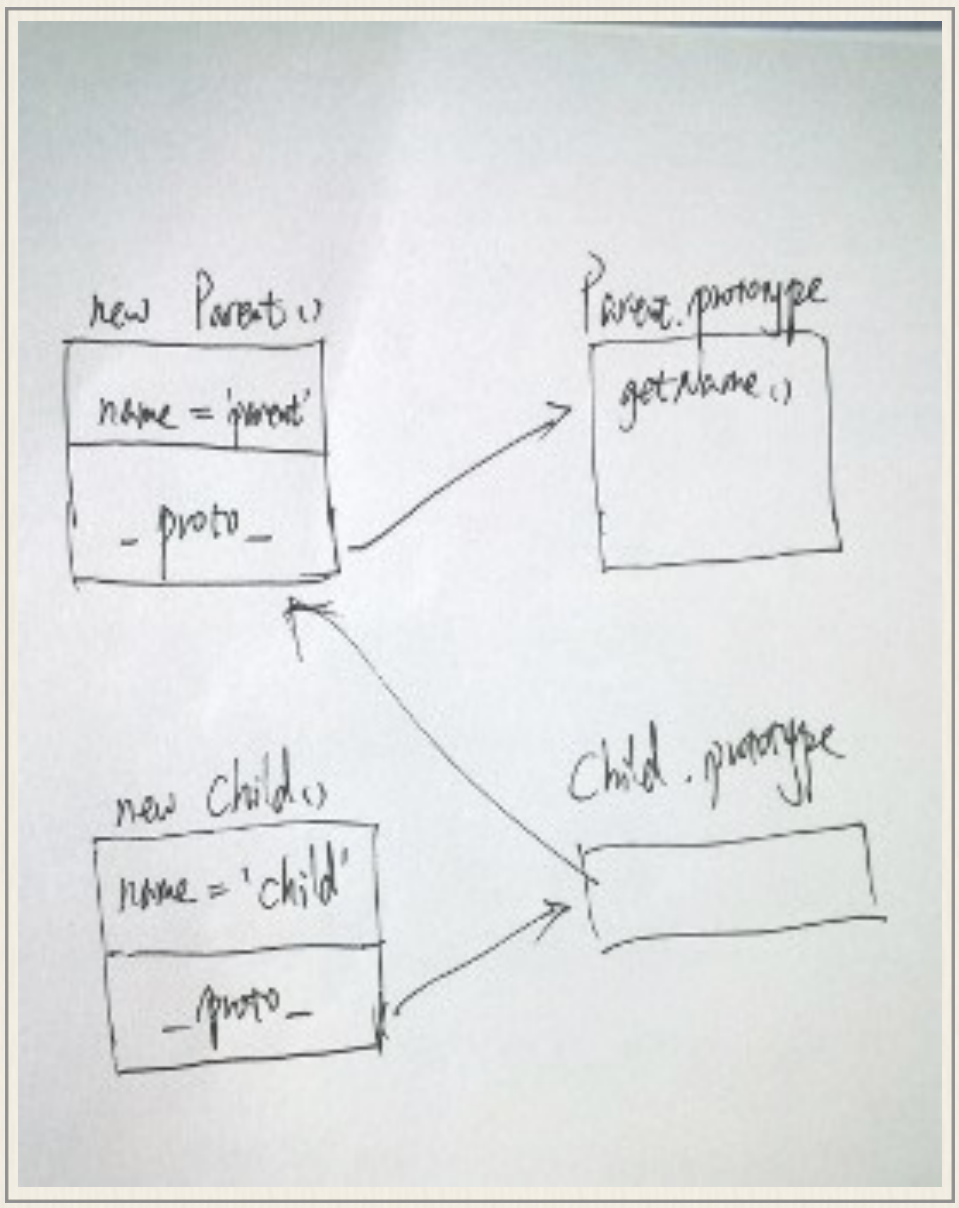
每个函数都是Function函数创建的对象，所以每个函数也有一个__proto__属性指向Function函数的原型。这里需要指出的是，真正形成原型链的是每个对象的__proto__属性，而不是函数的prototype属性，这是很重要的。

原型继承

基本模式

```
var Parent = function(){
    this.name = 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(){
    this.name = 'child' ;
} ;
Child.prototype = new Parent() ;

var parent = new Parent() ;
var child = new Child() ;

console.log(parent.getName()) ; //parent
console.log(child.getName()) ; //child
```

这种是最简单实现原型继承的方法，直接把父类的对象赋值给子类构造函数的原型，这样子类的对象就可以访问到父类以及父类构造函数的prototype中的属性。 这种方法的原型继承图如下：



这种方法的优点很明显，实现十分简单，不需要任何特殊的操作；同时缺点也很明显，如果子类需要做跟父类构造函数中相同的初始化动作，那么就得在子类构造函数中再重复一遍父类中的操作：

```
var Parent = function(name){
    this.name = name || 'parent' ;
} ;
Parent.prototype.getName = function(){
```

```
    return this.name ;
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(name){
    this.name = name || 'child' ;
} ;
Child.prototype = new Parent() ;

var parent = new Parent('myParent') ;
var child = new Child('myChild') ;

console.log(parent.getName()) ; //myParent
console.log(child.getName()) ; //myChild
```

上面这种情况还只是需要初始化name属性，如果初始化工作不断增加，这种方式是很不方便的。因此就有了下面一种改进的方式。

借用构造函数

```
var Parent = function(name){
    this.name = name || 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
```

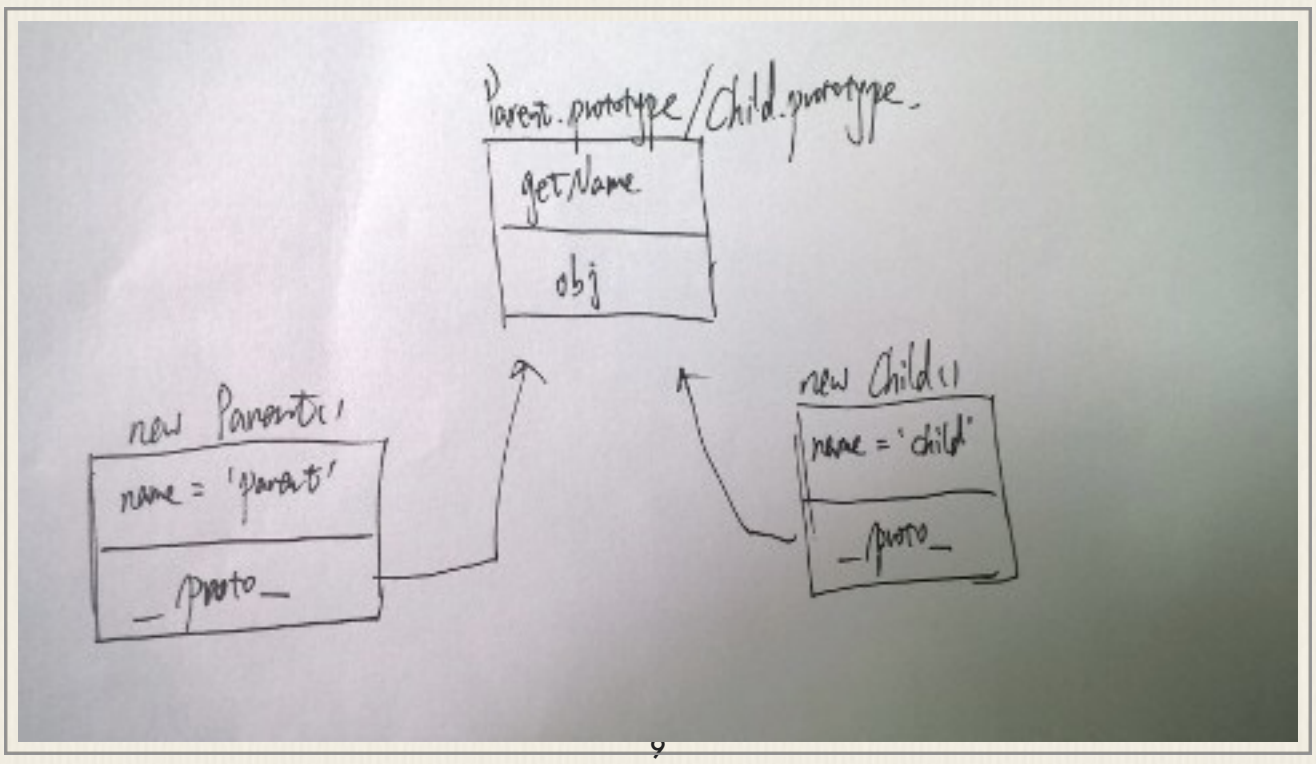
```
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(name){
    Parent.apply(this,arguments) ;
} ;
Child.prototype = new Parent() ;

var parent = new Parent('myParent') ;
var child = new Child('myChild') ;

console.log(parent.getName()) ; //myParent
console.log(child.getName()) ; //myChild
```

上面这种方法在子类构造函数中通过apply调用父类的构造函数来进行相同的初始化工作，这样不管父类中做了多少初始化工作，子类也可以执行同样的初始化工作。但是上面这种实现还存在一个问题，父类构造函数被执行了两次，一次是在子类构造函数中，一次在赋值子类原型时，这是很多余的，所以我们还需要做一个改进：

```
var Parent = function(name){
    this.name = name || 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
```

```
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(name){
    Parent.apply(this,arguments) ;
} ;
Child.prototype = Parent.prototype ;

var parent = new Parent('myParent') ;
var child = new Child('myChild') ;

console.log(parent.getName()) ; //myParent
console.log(child.getName()) ; //myChild
```

这样我们就只需要在子类构造函数中执行一次父类的构造函数，同时又可以继承父类原型中的属性，这也比较符合原型的初衷，就是把需要复用的内容放在原型中，我们也只是继承了原型中可复用的内容。上面这种方式的原型图如下：



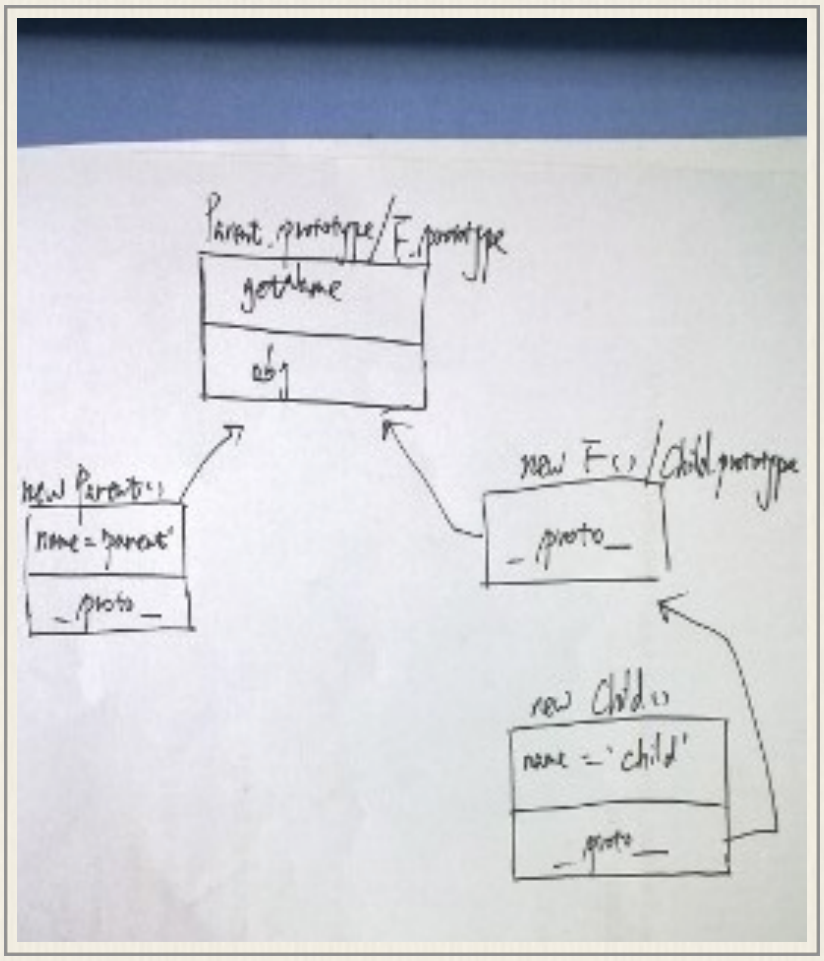
临时构造函数模式(圣杯模式)

上面借用构造函数模式最后改进的版本还是存在问题，它把父类的原型直接赋值给子类的原型，这就会造成一个问题，就是如果对子类的原型做了修改，那么 这个修改同时也会影响到父类的原型，进而影响父类对象，这个肯定不是大家所希望看到的。为了解决这个问题就有了临时构造函数模式。

```
var Parent = function(name){
    this.name = name || 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(name){
    Parent.apply(this,arguments) ;
} ;
var F = new Function(){} ;
F.prototype = Parent.prototype ;
Child.prototype = new F() ;

var parent = new Parent('myParent') ;
var child = new Child('myChild') ;

console.log(parent.getName()) ; //myParent
```

console.log(child.getName()) ; //myChild

该方法的原型继承图如下:



很容易可以看出，通过在父类原型和子类原型之间加入一个临时的构造函数F，切断了子类原型和父类原型之间的联系，这样当子类原型做修改时就不会影响到父类原型。

我的方法

《Javascript模式》中到圣杯模式就结束了，可是不管上面哪一种方法都有一个不容易被发现的问题。大家可以看到我在'Parent'的prototype属性中加入了一个obj对象字面量属性，但是一直都没有用。我们在圣杯模式的基础上来看看下面这种情况:

var Parent = function(name){

```
    this.name = name || 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(name){
    Parent.apply(this,arguments) ;
} ;
var F = new Function(){} ;
F.prototype = Parent.prototype ;
Child.prototype = new F() ;

var parent = new Parent('myParent') ;
var child = new Child('myChild') ;

console.log(child.obj.a) ; //1
console.log(parent.obj.a) ; //1
child.obj.a = 2 ;
console.log(child.obj.a) ; //2
console.log(parent.obj.a) ; //2
```

在上面这种情况中，当我修改child对象obj.a的时候，同时父类的原型中的obj.a也会被修改，这就发生了和共享原型同样的问题。出现这个情况是因为当访问child.obj.a的时候，我们会沿着原型链一直找到父类的prototype中，然后找到了obj属性，然后对obj.a进行修改。再看看下面这种情况：

```
var Parent = function(name){
    this.name = name || 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
} ;
Parent.prototype.obj = {a : 1} ;

var Child = function(name){
    Parent.apply(this,arguments) ;
} ;
var F = new Function(){} ;
F.prototype = Parent.prototype ;
Child.prototype = new F() ;

var parent = new Parent('myParent') ;
var child = new Child('myChild') ;

console.log(child.obj.a) ; //1
console.log(parent.obj.a) ; //1
```

```
child.obj.a = 2 ;
console.log(child.obj.a) ; //2
console.log(parent.obj.a) ; //2
```

这里有一个关键的问题，当对象访问原型中的属性时，原型中的属性对于对象来说是只读的，也就是说child对象可以读取obj对象，但是无法修改原型中obj对象引用，所以当child修改obj的时候并不会对原型中的obj产生影响，它只是在自身对象添加了一个obj属性，覆盖了父类原型中的obj属性。而当child对象修改obj.a时，它先读取了原型中obj的引用，这时候child.obj和Parent.prototype.obj是指向同一个对象的，所以child对obj.a的修改会影响到Parent.prototype.obj.a的值，进而影响父类的对象。AngularJS中关于$scope嵌套的继承方式就是模范Javasript中的原型继承来实现的。根据上面的描述，只要子类对象中访问到的原型跟父类原型是同一个对象，那么就会出现上面这种情况，所以我们可以对父类原型进行拷贝然后再赋值给子类原型，这样当子类修改原型中的属性时就只是修改父类原型的一个拷贝，并不会影响到父类原型。具体实现如下：

```
var deepClone = function(source,target){
    source = source || {} ;
    var toStr = Object.prototype.toString ,
        arrStr = '[object array]' ;
    for(var i in source){
        if(source.hasOwnProperty(i)){
            var item = source[i] ;
            if(typeof item === 'object'){
                target[i] = (toStr.apply(item).toLowerCase() === arrStr) : [] ? {}
;
```

```
                deepClone(item,target[i]) ;
            }else{
                deepClone(item,target[i]) ;
            }
        }
    }
    return target ;
} ;
var Parent = function(name){
    this.name = name || 'parent' ;
} ;
Parent.prototype.getName = function(){
    return this.name ;
} ;
Parent.prototype.obj = {a : '1'} ;

var Child = function(name){
    Parent.apply(this,arguments) ;
} ;
Child.prototype = deepClone(Parent.prototype) ;

var child = new Child('child') ;
var parent = new Parent('parent') ;
```

```
console.log(child.obj.a) ; //1
console.log(parent.obj.a) ; //1
child.obj.a = '2' ;
console.log(child.obj.a) ; //2
console.log(parent.obj.a) ; //1
```

综合上面所有的考虑，Javascript继承的具体实现如下，这里只考虑了Child和Parent都是函数的情况下：

```
var deepClone = function(source,target){
    source = source || {} ;
    var toStr = Object.prototype.toString ,
        arrStr = '[object array]' ;
    for(var i in source){
        if(source.hasOwnProperty(i)){
            var item = source[i] ;
            if(typeof item === 'object'){
                target[i] = (toStr.apply(item).toLowerCase() === arrStr) : [] ? {}
;
                deepClone(item,target[i]) ;
            }else{
                deepClone(item,target[i]) ;
            }
        }
    }
```

```
    return target ;
} ;

var extend = function(Parent,Child){
    Child = Child || function(){} ;
    if(Parent === undefined)
        return Child ;
    //借用父类构造函数
    Child = function(){
        Parent.apply(this,argument) ;
    } ;
    //通过深拷贝继承父类原型
    Child.prototype = deepClone(Parent.prototype) ;
    //重置constructor属性
    Child.prototype.constructor = Child ;
} ;
```

总结

说了这么多，其实Javascript中实现继承是十分灵活多样的，并没有一种最好的方法，需要根据不同的需求实现不同方式的继承，最重要的是要理解Javascript中实现继承的原理，也就是原型和原型链的问题，只要理解了这些，自己实现继承就可以游刃有余。

原文链接：http://segmentfault.com/blog/kk_470661/1190000000766541

Code Game 对技术的选取 兼谈为何不应该用 CoffeeScript 与 Less

作者：Zihua Li

这篇日志将以偏技术的角度介绍我最近在做的业余项目 Code Game，其中我会解释 Code Game 对某些技术是如何进行取舍的，包括为什么不使用 CoffeeScript 以及选择 Myth 代替 Less/Sass 的原因。

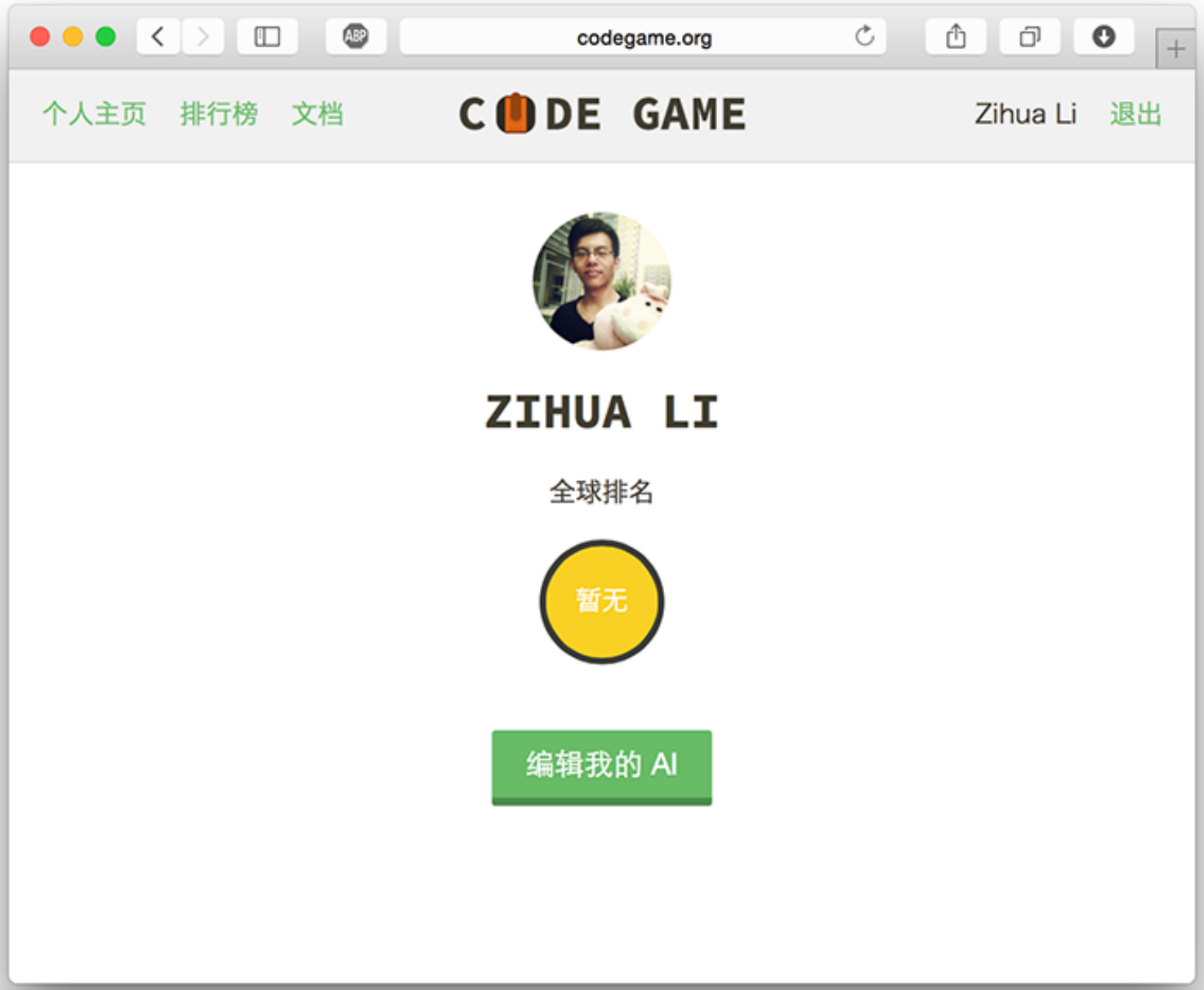
Code Game 是什么

Code Game 是我花费一个月的业余时间完成的 AI 脚本对战平台，网址是 http://codegame.org，玩家可以通过编写 JavaScript 脚本来控制游戏中的角色并与平台上的其他玩家进行竞赛。

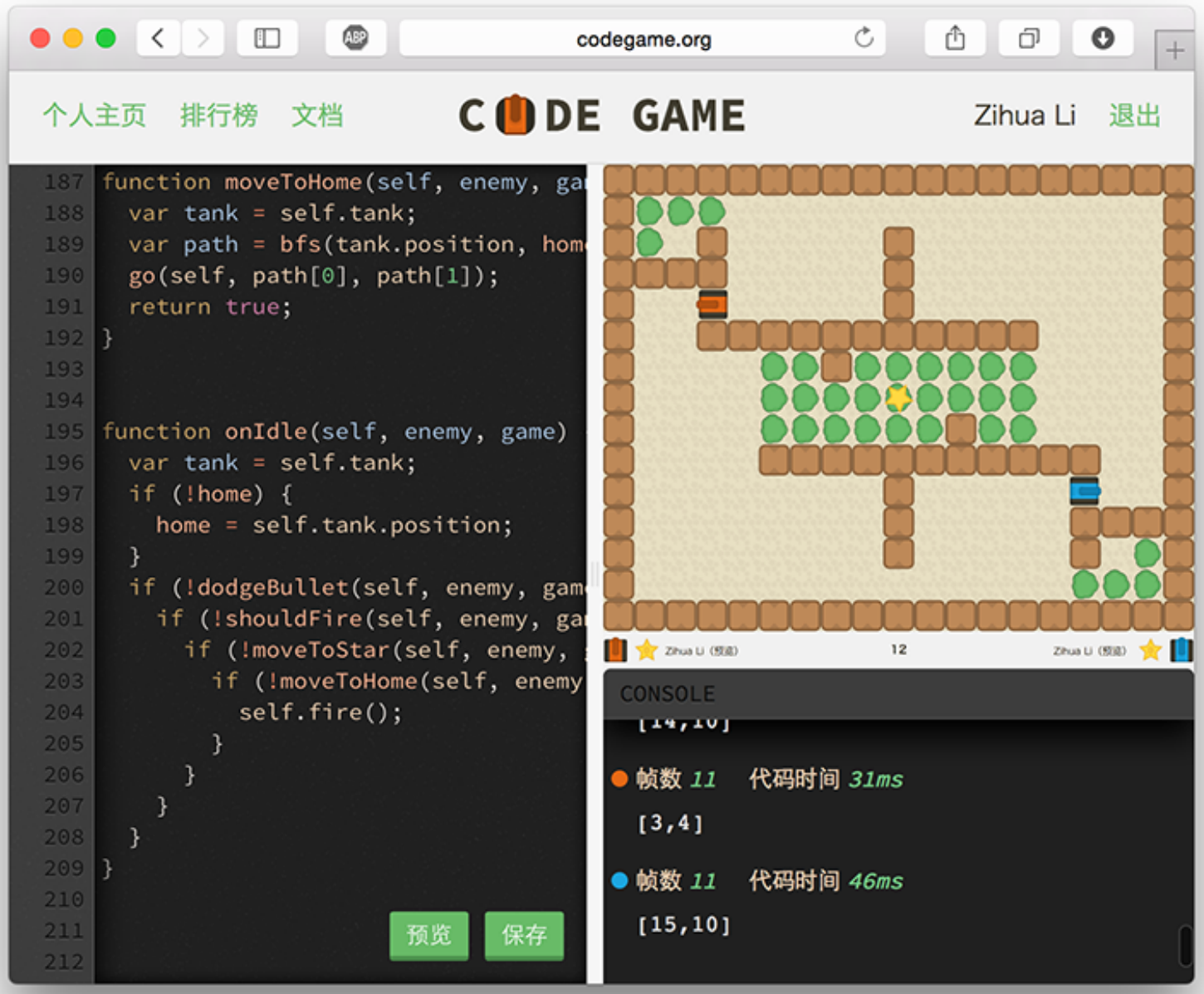
Code Game 的灵感来源于大学时我在北航 MSTC(Microsoft Technology Club) 参与的 BigTank 项目。BigTank 是一个使用 C# 开发的 3D 坦克对战游戏，与传统坦克对战游戏由玩家直接操控坦克不同，BigTank 的玩家需要通过编写 Lua 脚本来分析游戏局势并控制自己的坦克行动，本质上是一个关于 AI 算法的 Online Judge 平台。基于 BigTank，MSTC 举办了编程挑战赛，在北航获得了很好的反响。可惜限于开发周期和当时的技术水平，BigTank 本身存在很多不足，包括游戏规则欠考虑、脚本解析器存在不少 Bug 等，种种这些都或多或少地影响了比赛的精彩程度。一个月前，我终于决定重新开发一套类似 BigTank 的平台来弥补遗憾。

功能

Code Game 的页面构成很简单，挨个介绍也并不会花费很多篇幅。用户使用 GitHub 账号登录网站会进入个人主页页面，在这个页面中玩家可以看到自己的资料和 AI 的排名：

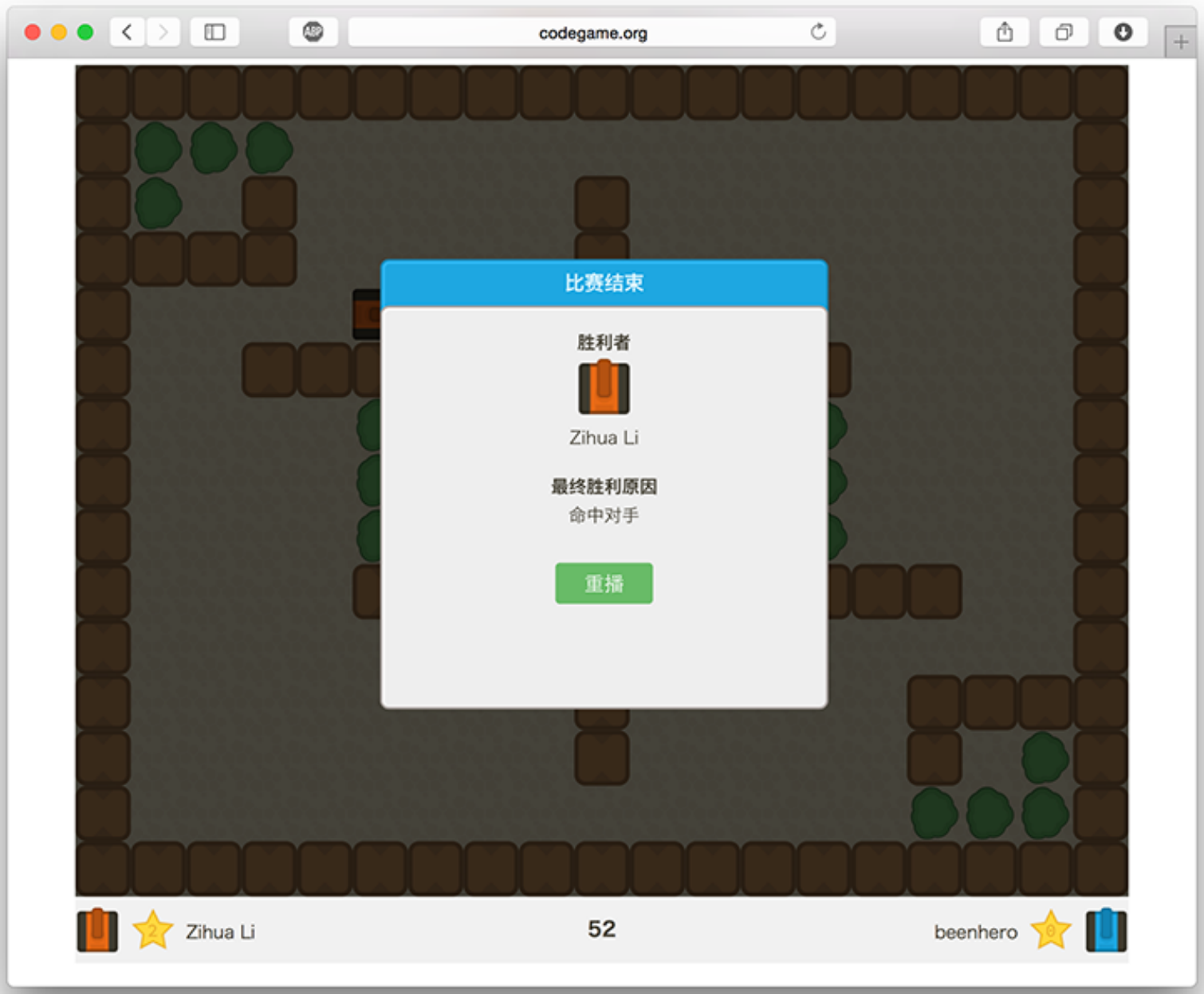


点击“编写我的 AI”按钮就来到了脚本编写页面，该页面由支持代码高亮和自动补全的在线编辑器、可以实时看到脚本的运行效果的预览画面以及用来查看调试信息的控制台组成：



Code Game 每天会根据 AI 的胜率更新一次排行榜，通过排行榜可以看到每个玩家的排名，并且点击昵称可以进入其个人主页：

玩家可以在他人的个人主页向对方发起挑战，发起挑战后玩家会进入挑战页面，并看到两个脚本的对战情况，游戏结束后页面会告知玩家胜者及胜利原因：



技术架构

Code Game 的技术架构以及对技术的取舍理念是优先考虑开发效率，并在保证运行效率可接受的前提下尽量降低技术栈复杂度。

1. 开发语言

Code Game 使用 Node.js 开发，除了我对其较熟悉以外，选择 Node.js 的另一个重要原因就是使用 Node.js 实现 JavaScript 脚本沙盒相较其他语言要容易得多。

2. 前端技术栈

前端使用了 Gulp + Myth + Browserify + Sketch-Tool，下面分别详细介绍。

1. Gulp

平常的开发中，对于小型项目我一般会使用 Make 或 NPM 的 scripts 来进行构建任务。而 Code Game 由于需要构建的内容较多，所以选择了 Gulp。相较于更为流行的 Grunt，Gulp 的 Stream 理念使得完成同样的构建任务时编写的代码更少。在 Code Game 中，Gulp 中的任务分为两类：一类任务用来实现检测源文件的改动并自动编译，比如将基于 Myth 编写的 CSS 实时编译为可以被浏览器解析的 CSS；另一类任务则用来执行生产环境的构建，比如合并 CSS、压缩 JavaScript 等。

2. Myth

Myth 在各种 CSS 预处理语言中绝对算不上流行，在 GitHub 上其共被 3000 余人 star，虽然不算少，但相比 Less 这样动辄一万多 star 的项目说是冷门也毫不过分。Myth 的优势和它的口号一样：“CSS the way it was imagined.” Myth 可以让你提前使用 CSS 的高级特性而无需考虑浏览器兼容问题。举例来说，当用到 transform 属性时通常还需要额外加上浏览器前缀 -webkit 来兼容 Safari 和 旧版的 Chrome，如果要兼容 IE 9，则更是要加上 -ms。而使用 Myth 则不用操心这个问题：只需要写一个 transform，Myth 会在编译过程中自动加上需要的前缀。Myth 与 Less、Sass 这样的预处理语言最大的区别就在于写 Less 时你是在写 Less，写 Sass 时你是在写 Sass，而当你写 Myth 时，你就是在写 CSS。这一点十分重要，因为把标准和草案都算上，CSS 语言本身已经足够完备了，它支持变量：

```
:root {
    --purple: #847AD1;
    --large: 10px;
}

a {
```

```
  color: var(--purple);
}

pre {
  padding: var(--large);
}
```

也支持数学计算：

```
pre {
  margin: calc(var(--large) * 2);
}
```

甚至还支持颜色处理：

```
a {
  color: var(--purple);
}

a:hover {
  color: color(var(--purple) tint(20%));
}
```

可以说需要用到的特性 CSS 本身就已经具备了，那么何必再使用另一种语言呢？更何况 Less 和 Sass 这样“强大”的预处理语言在带来开发上方便的同时也引入了很多问题，而大部分问题都可以归结到一点，即“你根本就不

是在写 CSS”。看下面的 Less 代码：

```
.container {
        width: 960px;
        overflow: hidden;
        .main {
          width: 61.8%;
          float: left;
          .post {
            background: #f00;
            .title {
              position: absolute;
              background: url("images/header-image.jpg");
            }
          }
        }
      }
```

Less 支持的样式嵌套很容易诱使开发者写出上面这样层层嵌套的代码，编译成 CSS 后，最长的 Selector 是.container .main .post .title，以纯 CSS 的眼光来看，应该减少嵌套层数来提高性能（比如改成.post .title），亦或是优化类名来实现样式模块化（比如把.post .title改成.post-title）。然 后一旦用 Less 写出来，就很难以 CSS 的角度来审视本就要编译成 CSS 的代码。很多使用 Less 或 Sass 的公司的 Style Guide 都会明确禁止过度嵌套，然而与其以规范来要求开发者，不如就单纯地使用 CSS，并享受 Myth 提供的便利来的方便自然。

3. Browserify

Browserify 可以非常方便地实现前端 JavaScript 的模块化。使用 Browserify，你可以在前端的 JavaScript 中使用和 Node.js 一样的模块加载方式，即 require('modules')， 使得前后端 JavaScript 模块级复用成为了可能。Code Game 游戏沙盒部分的所有模块曾经是前后端共用的，当用户在编辑器中预览时沙盒运行在前端，当与其他玩家竞赛时，沙盒则运行在后端。同时 Browserify 作为一个构建工具，并不影响前端脚本的加载逻辑，换言之在使用 Browserify 的同时依然可以使用 RequireJS、SeaJS 这样的 Module Loader 以及 Combo Handler 等技术。

除了 Browserify 以外，Code Game 没有使用其他的 JavaScript 预处理工具。也没有使用 CoffeeScript、LiveScript 这样的语言替代 JavaScript，原因在于 JavaScript 在前端工程方面本身已经足够优秀，而 CoffeeScript 和 LiveScript 这样的语言在提供更“现代”的语法同时，也会大大降低代码的可控性。同 Less 和 CSS 的关系一样，CoffeeScript 与 JavaScript 在语言层面的差异会导致二者间代码逻辑的不调和。一个最明显的例子是 CoffeeScript 提供的 class 关键词使得其可以像基于类Class-based的语言一样实现和管理对象，然而 JavaScript 本身却是是基于原型Prototype-based的语言，当一个以基于类的思想编写的代码逻辑编译成基于原型的语言时，其间产生的落差已经不是语言之间的优劣可以衡量的了。更何况 CoffeeScript 语言本身就存在太多的问题，比如 CoffeeScript 的函数调用无需写 ()，使得易读性大为下降（比较 console.log x + 1 和 console.log x +1），同时也引入了很多细节问题（比如无法实现 JavaScript 里的具名匿名函数 invoke(function func() {})），另外 CoffeeScript 生成的 JavaScript 虽然有很多最佳实践，但总体并不易读，也很容易生成冗余代码，比如：

```
greet = (name) ->
        for time in ['morning', 'afternoon', 'nignt']
          console.log "Good #{time}, #{name}!"

      greet 'Bob'
```

会生成：

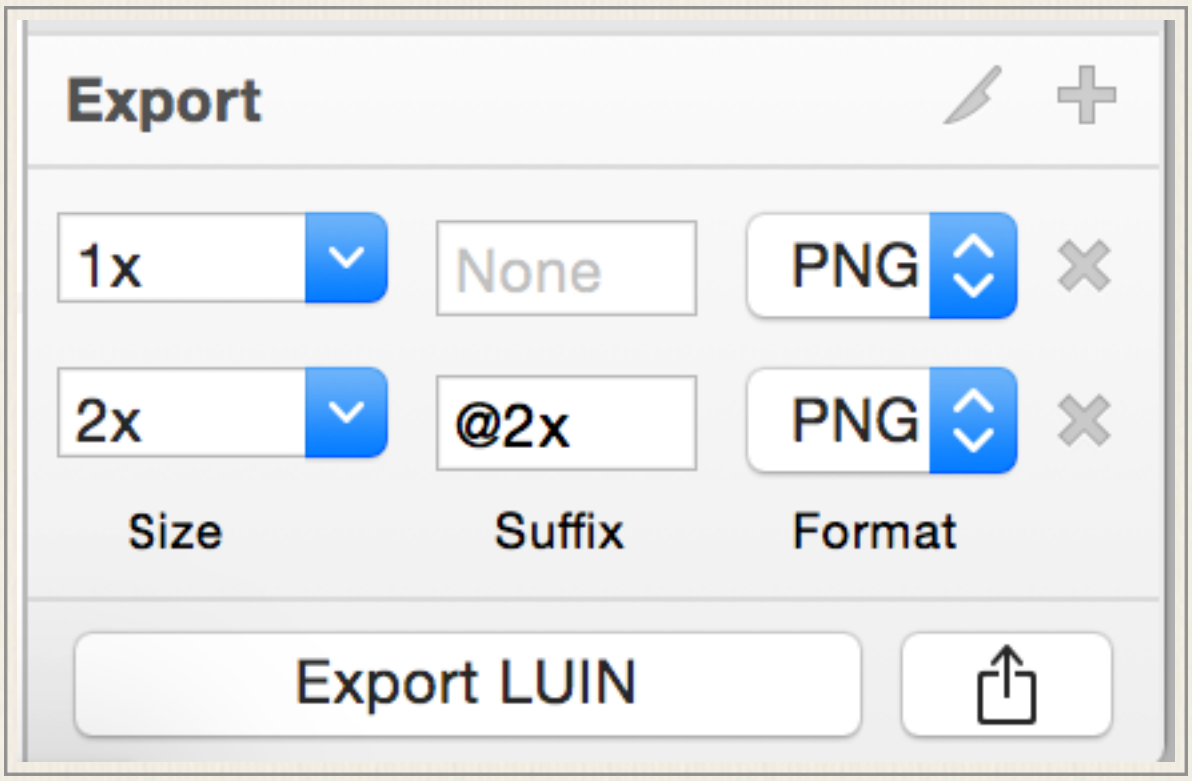
```
var greet;

    greet = function(name) {
      var time, _i, _len, _ref, _results;
      _ref = ['morning', 'afternoon', 'nignt'];
      _results = [];
      for (_i = 0, _len = _ref.length; _i < _len; _i++) {
        time = _ref[_i];
        _results.push(console.log("Good " + time + ", " + name + "!"));
      }
      return _results;
    };

    greet('Bob');
```

虽然语句表达式化是好的，免去写 return 也是好的，但是 CoffeeScript 终究还是一门要编译成 JavaScript 的语言，本来简单的代码变得如此复杂，即使表面再光鲜又有什么意义呢。CoffeeScript 对于初级 JavaScript 程序员来说，可以帮助他们避免很多 JavaScript 的陷阱，也能更顺畅地写出最佳实践的代码，但从整体而言，一个富有经验的 JavaScript 开发者写出的 JavaScript 更可能要比 CoffeeScript 的可控性来的高。

4. SketchTool

之前做设计一直使用 Photoshop，直到见识到 Sketch 的威力后便很少碰 Photoshop 了。网上有很多文章讨论两者的设计优劣，所以不再赘述。这里主要介绍对开发者来说 Sketch 的优势。

相较 Photoshop，Sketch 最大的优势就是可以实现切图自动化，对于每个图层来都可以指定其导出格式以及文件名：



更重要的是，Sketch 官方提供了命令行工具 SketchTool，可以通过命令将 Sketch 源文件按规则导出成图像文件，这意味着配合 Gulp 可以实现当修改 Sketch 的设计后自动切图。同时由于 Sketch 的文件普遍很小，所以甚至可以将其放入版本库中来维护其版本（自然这样也就无需将切好的图片放入版本库，因为这些图片可以由 Gulp 构建脚本生成）。

3. 沙盒的实现

在 Code Game 中最关键的一环就是沙盒的实现了。因为涉及到对战，所以比赛时双方选手的代码自然不能运行在前端以免让玩家看到对手的代码。所以 Code Game 采用如下流程来实现脚本对战：

1. 玩家在编辑器调试代码并保存

2. 服务端将玩家的代码保存到 MySQL 数据库中
3. 进行比赛时，服务端调集双方的代码，并在后端解析运行
4. 运行结束后，将游戏“录像”传回前端
5. 前端解析“录像”，并以动画形式展现给用户

下面分三个部分着重介绍 3 和 5 两个过程。

在后端解析玩家代码

这一过程是沙盒的意义所在。因为后端使用 Node.js 开发，而玩家的脚本本身是 JavaScript，所以解析脚本的过程本身就很简单，一个 eval() 即可。然而 eval() 并不能限制用户的脚本权限，从而使得用户的脚本可以访问 Node.js 的各种库函数，也会污染 Node.js 的全局变量，同时也无法对脚本的运行时间进行任何限制。

这个问题的解决方式是使用 Node.js 提供的 script.runInNewContext() 函数。runInNewContext() 函数接受两个参数：一个是全局变量对象，这个对象包含脚本可以使用的全部全局变量，在脚本中声明的全局变量也会保存在这个对象中；另一个是运行选项，可以在这个参数中指定脚本的超时时间，要注意的是这个参数是从 Node.js 0.11.x 开始支持的。

Code Game 定义了 Sandbox 类：

```
var Sandbox = module.exports = function(sandbox) {
  this.Math = Math;
  this.parseInt = parseInt;

  for (var key in sandbox) {
    if (sandbox.hasOwnProperty(key)) {
      this[key] = sandbox[key];
    }
```

```
  }
};
```

执行玩家脚本时，会实例化一个 Sandbox 实例作为全局变量对象，所以玩家能使用的全局变量只有 Math 和 parseInt。同时玩家的脚本都会声明一个 onIdle() 函数（具体可以参见官网文档），这个函数可以在之前的 Sandbox 实例 sandbox 中获取到。接下来需要实现在坦克空闲时执行 onIdle() 函数，如果直接调用 sandbox.onIdle() 来执行的话就无法借助 runInNewContext() 函数来实现超时检测了，所以 Code Game 使用如下的方法来解决这个问题：

```
Player.prototype.onIdle = function(self, enemy, game) {
  var code = 'onIdle(__self, __enemy, __game);';
  if (!this.script) {
    this.script = vm.createScript(code);
  }
  var start = Date.now();
  try {
    this.sandbox.__self = self;
    this.sandbox.__enemy = enemy;
    this.sandbox.__game = game;

    this.sandbox.print = function() {
      // ...
    };
    this.script.runInNewContext(this.sandbox, {
```

```
      timeout: 1500
    });
  } catch (e) {
    // ...
  }
  this.runTime += Date.now() - start;
};
```

首先为 __self、__enemy 和 __game 这三个不对用户公开的全局变量赋相应的值，之后使用同样的 Sandbox 实例执行代码 onIdle(__self, __enemy, __game)。因为调用了 runInNewContext()，所以可以定义超时规则。

录像文件的格式

为了将代码的执行结果在前端展示给用户，最重要的是把结果以一定规则记录下来，形成游戏录像。录像文件中记录了每个对象（坦克、子弹等）在每个帧的位置和执行的动作，如：

```
[
  [{
    "objectId": "6e723",
    "type": "tank",
    "direction": "right",
    "position": [6, 7],
    "action": "go",
  }, {
```

```
    "objectId": "4ad3f",
    "type": "tank",
    "direction": "left",
    "position": [10, 2],
    "action": "turn",
    "value": "left"
  }],
  [{
    "objectId": "6e723",
    "type": "tank",
    "direction": "right",
    "position": [6, 8],
    "action": "go"
  }],
]
```

首先录像的最外层是个数组，数组的一个元素代表一帧发生的所有动作。上面的示例录像中，第一帧坦克 6e723 从坐标 (6, 7) 向当前方向（右）前进了一个单位，同时坦克 4ad3f 向左转弯。第二帧坦克 6e723 从坐标 (6, 8) 继续前进了一步，同时坦克 4ad3f 没有执行任何操作。

理论上来说，录像文件中只要记录每个对象最初的状态和中间每步的动作即可使数据完整。然而可以注意到上面的录像样例中的每一帧都会把对象的所有信息 记录下来，包括朝向和坐标。这使得前端播放录像时可以从任意一帧开始播放，而不需要从头开始初始化对象状态，另外由于 Code Game 的录像一般都很小（游戏限定 200 帧之内必须结束），所以这样记录从成本考量也可以接受。

前端录像展示

一般而言在前端实现动画有如下几种方式：

1. 通过 JavaScript 操作 DOM
2. 使用 CSS Animation
3. Canvas 动画

在开发 Code Game 项目时，首先排除的是 Canvas。虽然 Canvas 的性能优异且兼容性良好，但是就播放录像这样简单的需求而言使用 Canvas 开发相对繁琐。其次排除的是直接操作 DOM，因为 JavaScript 实现的动画进行微小的位移时会出现抖动，而 Code Game 开发时希望对战页面可以在移动设备上播放，同时用户可以自定义播放速度，这就使得小位移的动画非常容易出现。

所以最后采取的播放录像的动画方案是 CSS Animation。录像中的每个拥有 objectId 属性的对象都会为其生成 DOM 节点，节点的 ID 由 objectid 构成，同时根据 type 的不同为其赋予不同的背景图。具体的动画实现以下面的动作为例：

```
{
  "objectId": "6e723",
  "type": "tank",
  "direction": "right",
  "position": [6, 7],
  "action": "go",
}
```

首先通过 JavaScript 找到 6e723 DOM 节点，然后根据当前游戏的 FPS 和地图大小 修改该节点的 Transition 时间，同时通过 transform 的 translate 操作来移动对象。

这样的实现存在一个问题，假设一个坦克不转向，一直前进了 10 个单位，前端会修改 10 次节点的 transition 和 transform 属性，每次只移动一个单位。而实际上对于这种情况可以优化成只修改一次属性，一次直接前进 10 个单位，当然 transition 的时间也要相应乘以 10。因为在游戏中坦克直线连续行进的地方很多，所以这种优化效果很明显。为此 Code Game 在前端会在解析录像前对录像进行优化，合并直线前进的操作。如上面的录像实例会被优化成：

```
[
  [{
    "objectId": "6e723",
    "type": "tank",
    "direction": "right",
    "position": [6, 7],
    "action": "go",
    "frames": 2
  }, {
    "objectId": "4ad3f",
    "type": "tank",
    "direction": "left",
    "position": [10, 2],
    "action": "turn",
    "value": "left"
  }],
  [],
]
```

坦克 6e723 在第一帧的动作中增加了 frames: 2 这个属性，表明一共前进了两帧，在之后播放动画时就可以大大降低修改 DOM 属性的次数了。

结语

这篇文章从开发 Code Game 项目的角度介绍了我对 Less/Sass 和 CoffeeScript 这些流行技术的取舍，以及在 Node.js 中开发沙盒的一些经验。然而技术本身其实并没有明显的优劣可言，只有基于特定项目和特定的开发者讨论时，关于“取舍”的话题才有意义。如果大家关于本文有什么想法，欢迎留言讨论。

另外 Code Game 已经在 GitHub 上开源，欢迎 Star！

原文链接：http://zihua.li/2014/11/talk-about-codegame/?utm_source=tuicool

JVM堆大小的自适应能力

译者：有孚

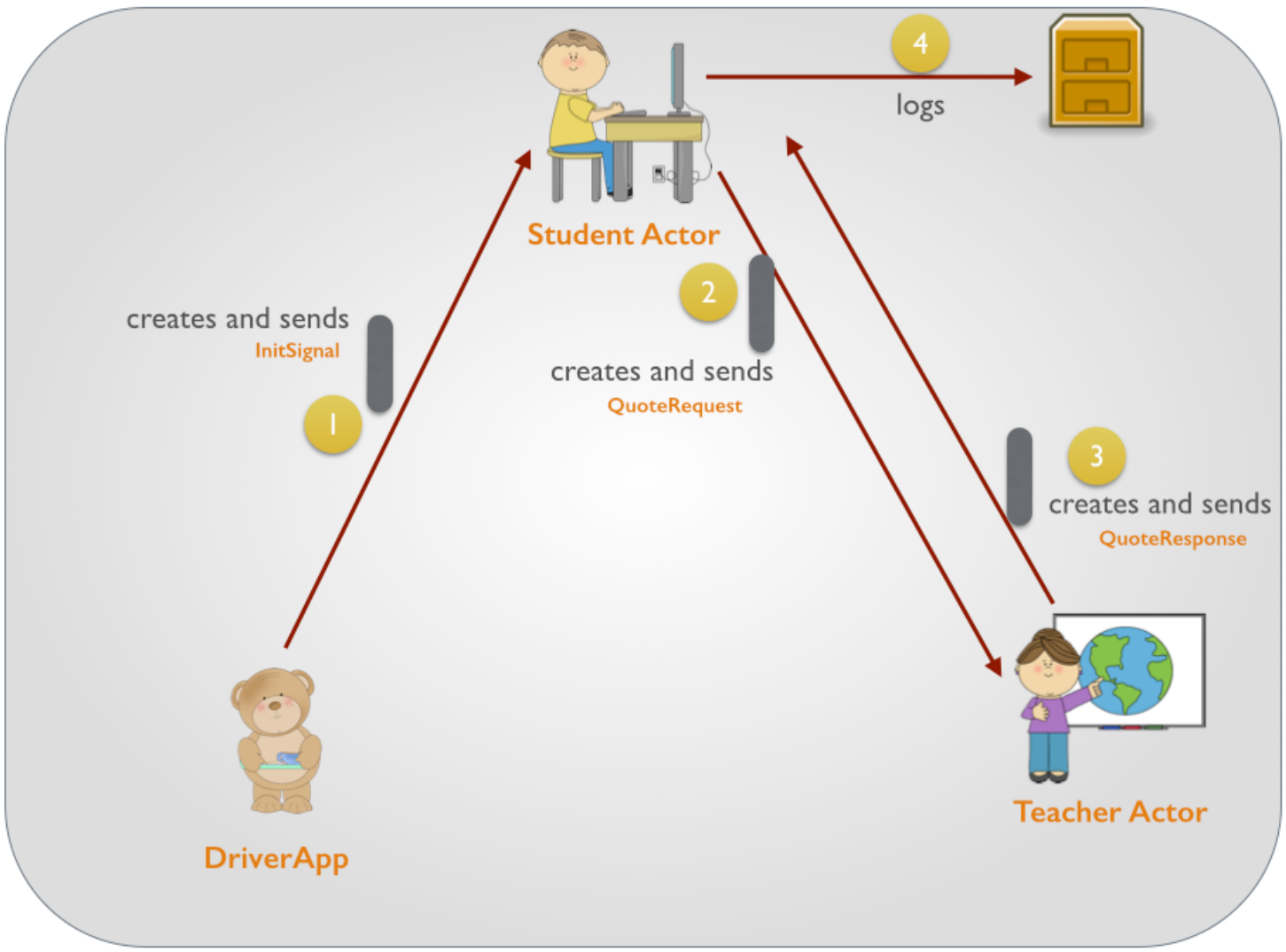
前面我们讲到了Actor的消息传递，并看到了如何发送一条fire-n-forget消息（也就是说，消息发送给Actor后我们就不管了，不从Actor那接收响应）。

技术上来讲，消息发送给Actor就是希望能有副作用的。设计上便是如此。目标Actor可以不做响应，也可以做如下两件事情——

1. 给发送方回复一条响应（在本例中，TeacherActor会将一句名言回复给StudentActor）
2. 将响应转发给其它的目标受众Actor，后者也可以进行响应/转发/产生副作用。Router和Supervisor就是这种情况。（很快我们就会看到）

请求及响应

本文中我们只关注第一点——请求及响应周期。



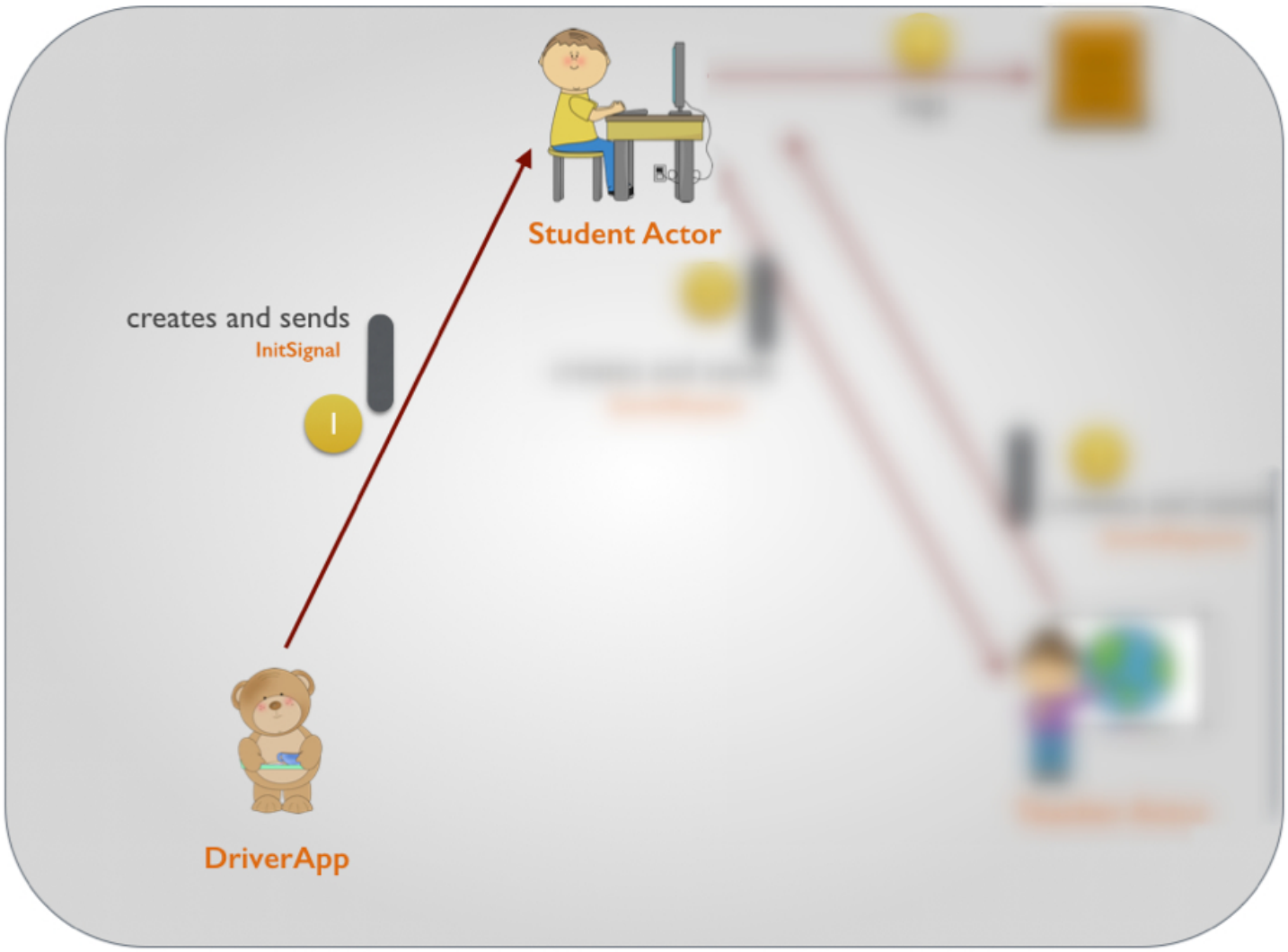
这张图说明了我们这次要做的事情。为了简单点，图中我并没有画出ActorSystem, Dispatcher以及Mailbox。

1. DriverApp将一条InitSignal消息发送给StudentActor。
2. StudentActor响应InitSignal消息并将一条QuoteRequest消息发送到TeacherActor。
3. 正如前面所说的那样，TeacherActor会回复一个QuoteResponse。
4. StudentActor将日志打印到控制台或者logger里。

同样的，我们会写一个测试用例来验证下它。

现在我们来仔细地分析下这四个步骤：

1. DRIVERAPP将一条INITSIGNAL消息发送给STUDENTACTOR



现在你应该能猜到DriverApp到底是干什么的了。它只做了4件事情:

1. 初始化ActorSystem

```
//Initialize the ActorSystem
  val system = ActorSystem("UniversityMessageSystem”)
```

2. 创建TeacherActor

```
//create the teacher actor
```

val teacherRef = system.actorOf(Props[TeacherActor], "teacherActor”)

3. 创建StudentActor

//create the Student Actor - pass the teacher actorref as a constructor parameter to StudentActor

val studentRef = system.actorOf(Props(new StudentActor(teacher-Ref)), "studentActor")

你会注意到我把TeacherActor的一个ActorRef的引用作为构造函数的参数传给了StudentActor，这样StudentActor才能够通过ActorRef来将消息发送给TeacherActor。当然还有别的方法（比如通过Props来传递），不过这么做对后续即将讲到的Supervisor和Router来说会方便一点。很快我们会看到子Actor也能实现这个功能，不过那个方法用在这里并不适合——学生来生成老师，这看起来不太对劲吧?

最后，

4. DriverApp将InitSignal消息发送给了StudentActor，这样StudentActor会开始将QuoteRequest消息发送给TeacherActor。

//send a message to the Student Actor

studentRef ! InitSignal

DriverClass讲的已经够多了。后面的Thread.sleep和ActorSystem.shutdown就是等了几秒，以便消息发送完成，然后再最终将ActorSystem关掉。

DRIVERAPP.SCALA

```
package me.rerun.akkanotes.messaging.requestresponse

import akka.actor.ActorSystem
import akka.actor.Props
import me.rerun.akkanotes.messaging.protocols.StudentProtocol._
import akka.actor.ActorRef

object DriverApp extends App {

  //Initialize the ActorSystem
  val system = ActorSystem("UniversityMessageSystem")

      //construct the teacher actor
      val teacherRef = system.actorOf(Props[TeacherActor], "teacherActor")

      //construct the Student Actor - pass the teacher actorref as a construc-
tor parameter to StudentActor
      val studentRef = system.actorOf(Props(new StudentActor(teacher-
Ref)), "studentActor")

      //send a message to the Student Actor
      studentRef ! InitSignal

      //Let's wait for a couple of seconds before we shut down the system
      Thread.sleep(2000)
```

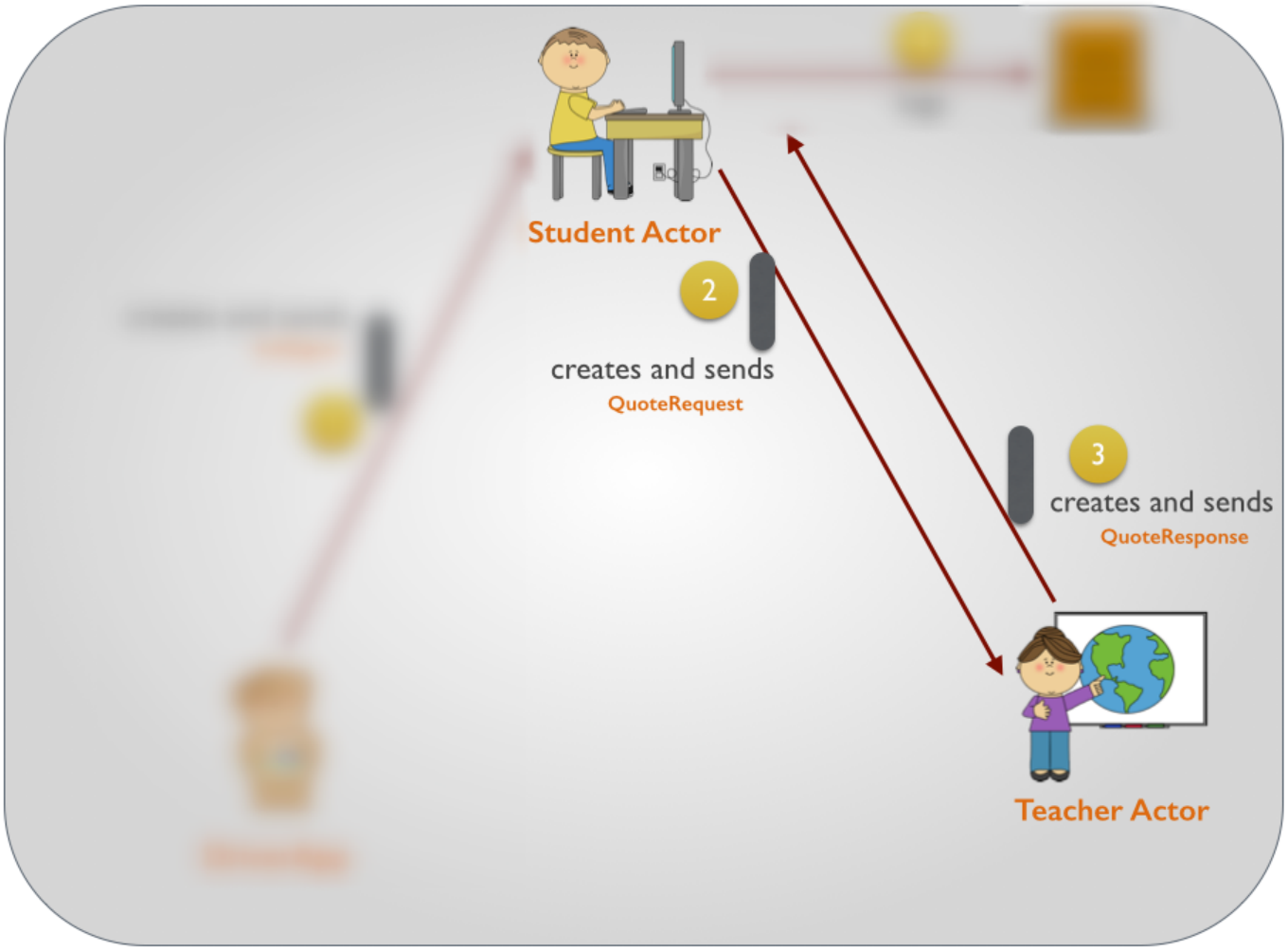
```
    //Shut down the ActorSystem.
    system.shutdown()

  }
```

2. STUDENTACTOR响应INITSIGNAL消息并将QUOTEREQUEST消息发送给TEACHERACTOR

以及

4. STUDENTACTOR接收到TEACHERACTOR回复的QuoteResponse然后将日志打印到控制台/logger上来

为什么我把第2和第4点放到一起来讲？因为它太简单了，如果分开讲的话我怕你嫌我啰嗦。

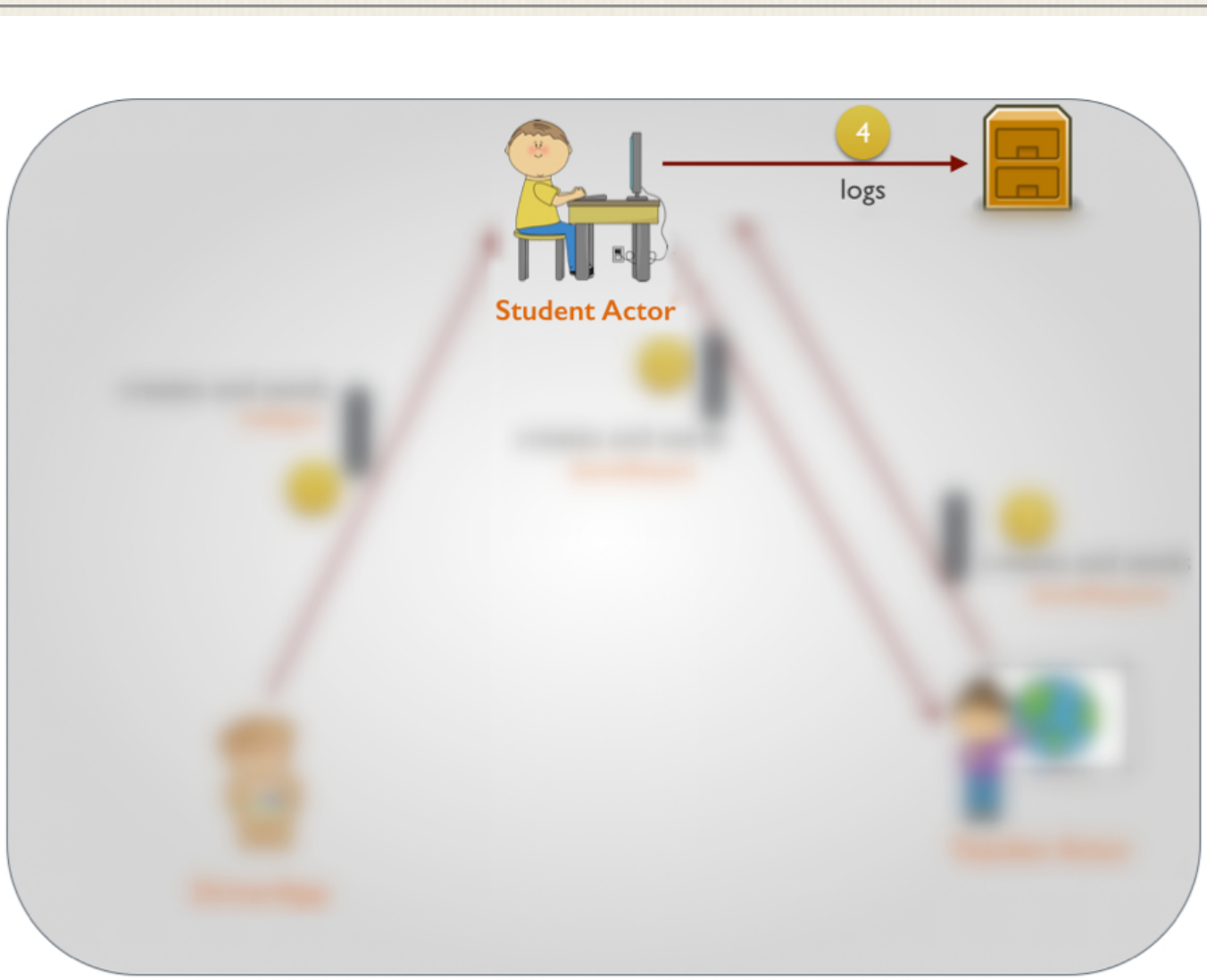


那么，第2步——StudentActor接收到DriverApp发过来的InitSingal消息并将QuoteRequest发送给TeacherActor。

```
def receive = {
    case InitSignal=> {
        teacherActorRef!QuoteRequest
    }
    ...
    ...
```

搞定！

第4步——StudentActor将TeacherActor发过来的消息打印出来。



说到做到：

```
case QuoteResponse(quoteString) => {
    log.info ("Received QuoteResponse from Teacher")
    log.info(s"Printing from Student Actor $quoteString")
}
```

我猜你肯定觉得这很像是伪代码。

那么，完整的StudentActor应该是这样的：

STUDENTACTOR.SCALA

```
package me.rerun.akkanotes.messaging.requestresponse

import akka.actor.Actor
import akka.actor.ActorLogging
import me.rerun.akkanotes.messaging.protocols.TeacherProtocol._
import me.rerun.akkanotes.messaging.protocols.StudentProtocol._
import akka.actor.Props
import akka.actor.ActorRef

class StudentActor (teacherActorRef:ActorRef) extends Actor with Actor-
Logging {
```

```
  def receive = {
    case InitSignal=> {
      teacherActorRef!QuoteRequest
    }

    case QuoteResponse(quoteString) => {
      log.info ("Received QuoteResponse from Teacher")
      log.info(s"Printing from Student Actor $quoteString")
    }
  }
}
```

3. TeacherActor回复QuoteResponse

这和我们在前面的fire-n-forget那篇)中看到的代码是类似的。

TeacherActor接收到QuoteRequest消息然后回复一个QuoteResponse。

TEACHERACTOR.SCALA

```
package me.rerun.akkanotes.messaging.requestresponse

import scala.util.Random

import akka.actor.Actor
import akka.actor.ActorLogging
import akka.actor.actorRef2Scala
```

```
import me.rerun.akkanotes.messaging.protocols.TeacherProtocol._

class TeacherActor extends Actor with ActorLogging {

  val quotes = List(
    "Moderation is for cowards",
    "Anything worth doing is worth overdoing",
    "The trouble is you think you have time",
    "You never gonna know if you never even try")

  def receive = {

      case QuoteRequest => {

        import util.Random

        //Get a random Quote from the list and construct a response
        val quoteResponse =
QuoteResponse(quotes(Random.nextInt(quotes.size)))

        //respond back to the Student who is the original sender of QuoteRe-
quest
        sender ! quoteResponse

      }
```

```
  }
}
```

测试用例

现在，我们的测试用例会来模拟下DriverApp。由于StudentActor只是打印了个日志消息，我们没法对QuoteResponse本身进行断言，那么我们就看下EventStream中是不是有这条日志消息就好了（就像上回做的那样）

那么，我们的测试用例看起来会是这样的：

```
"A student" must {

  "log a QuoteResponse eventually when an InitSignal is sent to it" in {

    import me.rerun.akkanotes.messaging.protocols.StudentProtocol._

    val teacherRef = system.actorOf(Props[TeacherActor], "teacherActor")

    val studentRef = system.actorOf(Props(new StudentActor(teacherRef)), "studentActor")


    EventFilter.info (start="Printing from Student Actor", occurrences=1).intercept{
      studentRef!InitSignal
    }
  }
}
```

代码

项目的完整代码可以从Github中进行下载。

原译文链接：http://it.deepinmind.com/gc/2014/11/07/adaptive-heap-sizing.html

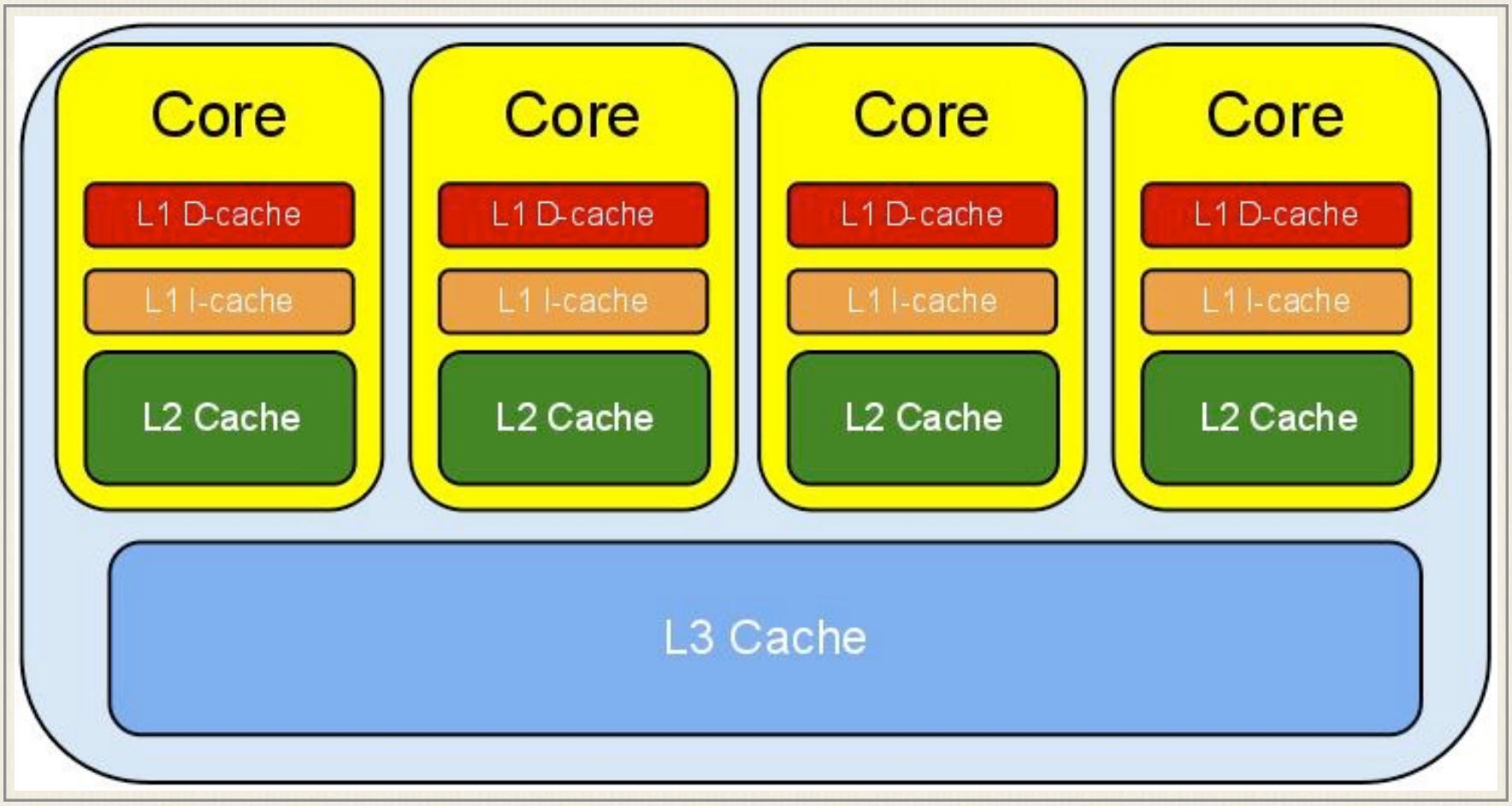
原文链接：http://rerun.me/2014/10/06/akka-notes-actor-messaging-request-and-response-3/

写Java也得了解CPU--CPU缓存

作者：macemers

CPU，一般认为写C/C++的才需要了解，写高级语言的(Java/C#/pathon...)并不需要了解那么底层的东西。我一开始也是这么想的，但直到碰到LMAX的Disruptor，以及马丁的博文，才发现写Java的，更加不能忽视CPU。经过一段时间的阅读，希望总结一下自己的阅读后的感悟。本文主要谈谈CPU缓存对Java编程的影响，不涉及具体CPU缓存的机制和实现。

现代CPU的缓存结构一般分三层，L1，L2和L3。如下图所示：



级别越小的缓存，越接近CPU， 意味着速度越快且容量越少。

L1是最接近CPU的，它容量最小，速度最快，每个核上都有一个L1 Cache(准确地说每个核上有两个L1 Cache， 一个存数据 L1d Cache， 一个存指令 L1i Cache)；

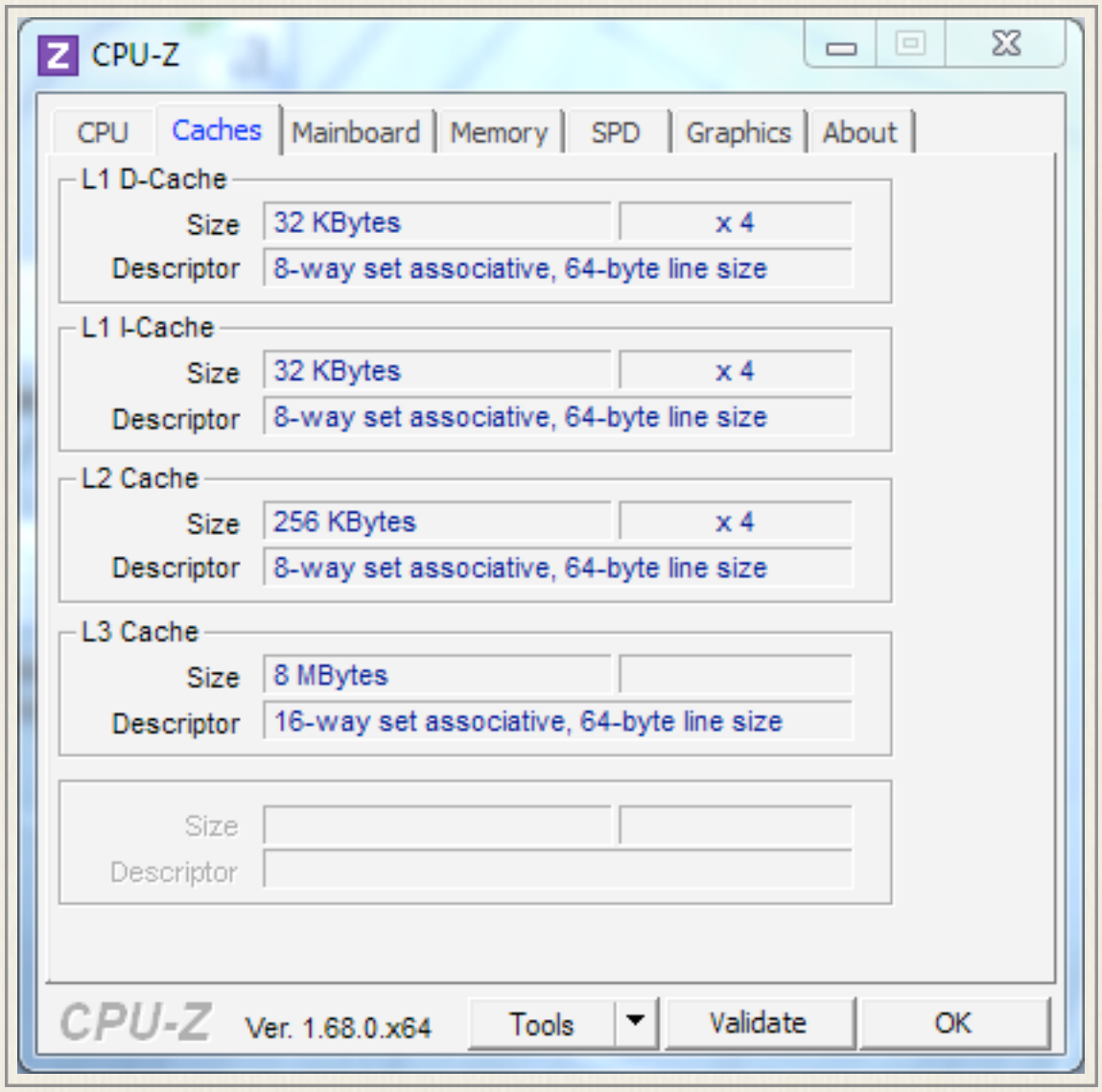
L2 Cache 更大一些，例如256K，速度要慢一些，一般情况下每个核上都有一个独立的L2 Cache；

L3 Cache是三级缓存中最大的一级，例如12MB，同时也是最慢的一级，在同一个CPU插槽之间的核共享一个L3 Cache。

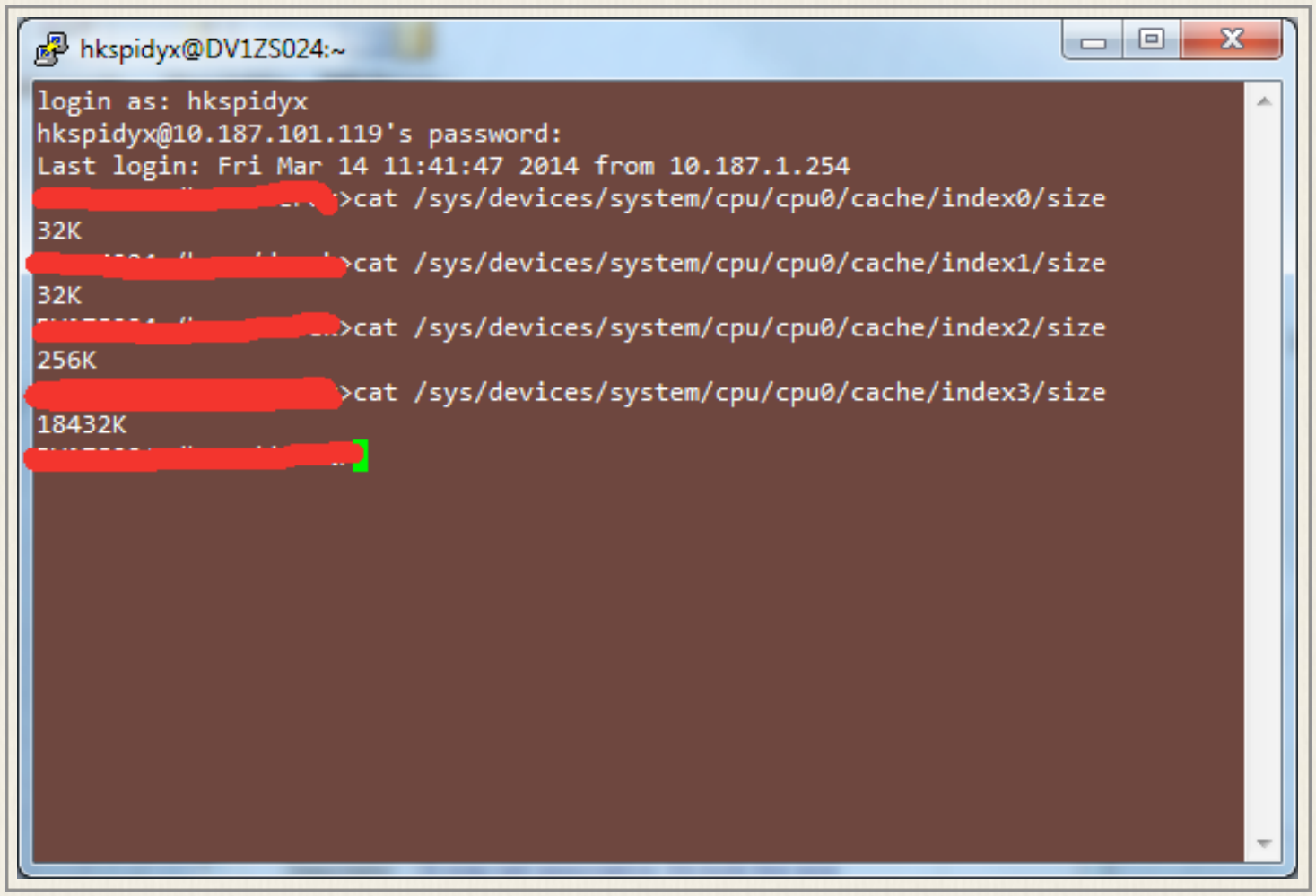
当CPU运作时，它首先去L1寻找它所需要的数据，然后去L2，然后去L3。如果三级缓存都没找到它需要的数据，则从内存里获取数据。寻找的路径越 长，耗时越长。所以如果要非常频繁的获取某些数据，保证这些数据在L1缓存里。这样速度将非常快。下表表示了CPU到各缓存和内存之间的大概速度：

从CPU到	大约需要的CPU周期	大约需要的时间(单位ns)
寄存器	1 cycle	
L1 Cache	~3-4 cycles	~0.5-1 ns
L2 Cache	~10-20 cycles	~3-7 ns
L3 Cache	~40-45 cycles	~15 ns
跨槽传输		~20 ns
内存	~120-240 cycles	~60-120ns

利用CPU-Z可以查看CPU缓存的信息：



在linux下可以使用下列命令查看：



有了上面对CPU的大概了解，我们来看看缓存行(Cache line)。缓存，是由缓存行组成的。一般一行缓存行有64字节(由上图"64-byte line size"可知)。所以使用缓存时，并不是一个一个字节使用，而是一行缓存行、一行缓存行这样使用；换句话说，CPU存取缓存都是按照一行，为最小单位操作的。

这意味着，如果没有好好利用缓存行的话，程序可能会遇到性能的问题。可看下面的程序：

```
1 public class L1CacheMiss {
2     private static final int RUNS = 10;
3     private static final int DIMENSION_1 = 1024 * 1024;
4     private static final int DIMENSION_2 = 6;
5
6     private static long[][] longs;
7
8     public static void main(String[] args) throws Exception {
9         Thread.sleep(10000);
10        longs = new long[DIMENSION_1][];
11        for (int i = 0; i < DIMENSION_1; i++) {
12            longs[i] = new long[DIMENSION_2];
13            for (int j = 0; j < DIMENSION_2; j++) {
14                longs[i][j] = 0L;
15            }
16        }
```

```
17        System.out.println("starting....");
18
19        long sum = 0L;
20        for (int r = 0; r < RUNS; r++) {
21
22            final long start = System.nanoTime();
23
24            //slow
25 //           for (int j = 0; j < DIMENSION_2; j++) {
26 //               for (int i = 0; i < DIMENSION_1; i++) {
27 //                   sum += longs[i][j];
28 //               }
29 //           }
30
31            //fast
32            for (int i = 0; i < DIMENSION_1; i++) {
33                for (int j = 0; j < DIMENSION_2; j++) {
34                    sum += longs[i][j];
35                }
36            }
37
38            System.out.println((System.nanoTime() - start));
39        }
40
```

```
41    }
42}
}
```

以我所使用的Xeon E3 CPU和64位操作系统和64位JVM为例，如这里所说，假设编译器采用行主序存储数组。

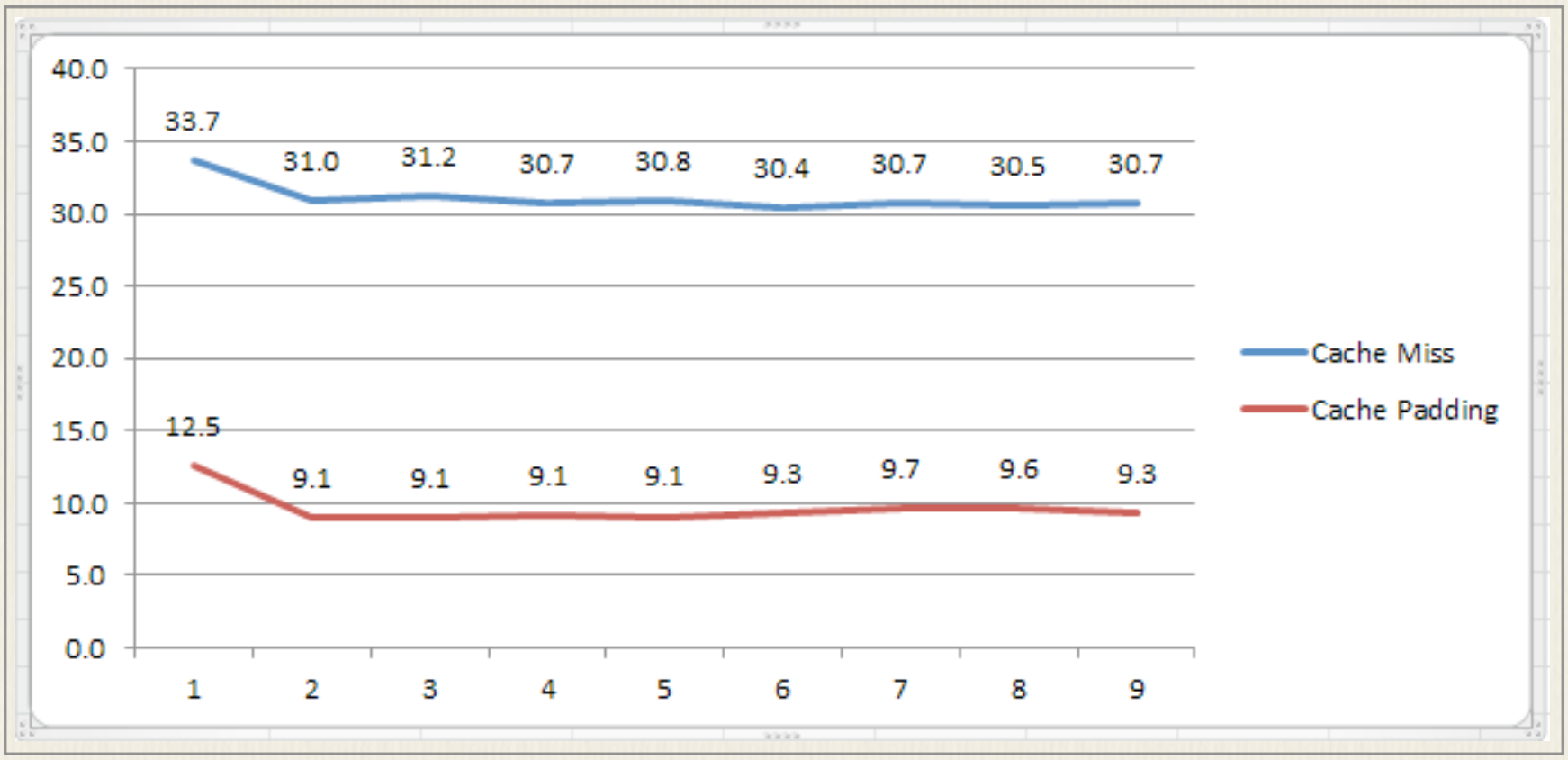
64位系统，Java数组对象头固定占16字节（未证实），而long类型占8个字节。所以16+8*6=64字节，刚好等于一条缓存行的长度：

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
long[6] @ 0x7d5d6cc58	64	64
long[6] @ 0x7d5d6cc18	64	64
long[6] @ 0x7d5d6cbd8	64	64
long[6] @ 0x7d5d6cb98	64	64
long[6] @ 0x7d5d6cb58	64	64
long[6] @ 0x7d5d6cb18	64	64
long[6] @ 0x7d5d6cad8	64	64
long[6] @ 0x7d5d6ca98	64	64
long[6] @ 0x7d5d6ca58	64	64
long[6] @ 0x7d5d6ca18	64	64
long[6] @ 0x7d5d6c9d8	64	64
long[6] @ 0x7d5d6c998	64	64
long[6] @ 0x7d5d6c958	64	64
long[6] @ 0x7d5d6c918	64	64
long[6] @ 0x7d5d6c8d8	64	64
long[6] @ 0x7d5d6c898	64	64
long[6] @ 0x7d5d6c858	64	64
long[6] @ 0x7d5d6c818	64	64
long[6] @ 0x7d5d6c7d8	64	64
long[6] @ 0x7d5d6c798	64	64
long[6] @ 0x7d5d6c758	64	64
long[6] @ 0x7d5d6c718	64	64
long[6] @ 0x7d5d6c6d8	64	64
long[6] @ 0x7d5d6c698	64	64
long[6] @ 0x7d5d6c658	64	64
long[6] @ 0x7d5d6c618	64	64
long[6] @ 0x7d5d6c5d8	64	64
long[6] @ 0x7d5d6c598	64	64
long[6] @ 0x7d5d6c558	64	64

如32-36行代码所示，每次开始内循环时，从内存抓取的数据块实际上覆盖了longs[i][0]到longs[i][5]的全部数据（刚好64字节）。因此，内循环时所有的数据都在L1缓存可以命中，遍历将非常快。

假如，将32-36行代码注释而用25-29 行代码代替，那么将会造成大量的缓存失效。因为每次从内存抓取的都是同行不同列的数据块（如longs[i][0]到longs[i][5]的全部数据），但循环下一个的目标，却是同列不同行（如longs[0][0]下一个是longs[1] [0]，造成了longs[0][1]-longs[0][5]无法重复利用）。运行时间的差距如下图，单位是微秒(us):

最后，我们都希望需要的数据都在L1缓存里，但事实上经常事与愿违，所以缓存失效 (Cache Miss)是常有的事，也是我们需要避免的事。



一般来说，缓存失效有三种情况:

1. 第一次访问数据, 在cache中根本不存在这条数据, 所以cache miss, 可以通过prefetch解决。

2. cache冲突, 需要通过补齐来解决（伪共享的产生）。

3. cache满, 一般情况下我们需要减少操作的数据大小, 尽量按数据的物理顺序访问数据。

原文链接：http://www.cnblogs.com/techyc/p/3607085.html

内存计算技术那家强？SPARK vs HANA

作者：吴朱华

最近业界有很多技术和产品都认为属于内存计算的范畴，由于我个人也从事于内存计算产品的研发，所以想借个机会，跟各位聊聊到底什么是内存计算技术，以及比较一些现在两种比较主流的内存计算技术Apache Spark和SAP HANA，它们的特点和区别。

什么是内存计算技术?

关于内存计算，就像云计算和大数据一样，其实无论在百度百科还是Wikipedia都没有非常精确的描述，但是有几个共通的关键点，我在这里给大家 总结一下：其一是数据放在内存中，至少和当前查询工作涉及到的数据放在都要放在内存中；其二是多线程和多机并行，也就是尽可能地利用现代x86 Xeon CPU线程数多的优势来加速整个查询；其三是支持多种类型的工作负载，除了常见和基本的SQL查询之后，还通常支持数据挖掘，更有甚者支持Full Stack（全栈），也就是常见编程模型都要支持，比如说SQL查询，流计算和数据挖掘等。

Apache Spark的设计思路

大家都知道，现在Apache Spark可以说是最火的开源大数据项目，就连EMC旗下专门做大数据Pivotal也开始抛弃其自研十几年GreenPlum技术，转而投入到 Spark技术开发当中，并且从整个业界而言，Spark火的程度也只有IaaS界的OpenStack能相提并论。那么本文作为一篇技术文章，我们接着 就直接切入它的核心机制吧。

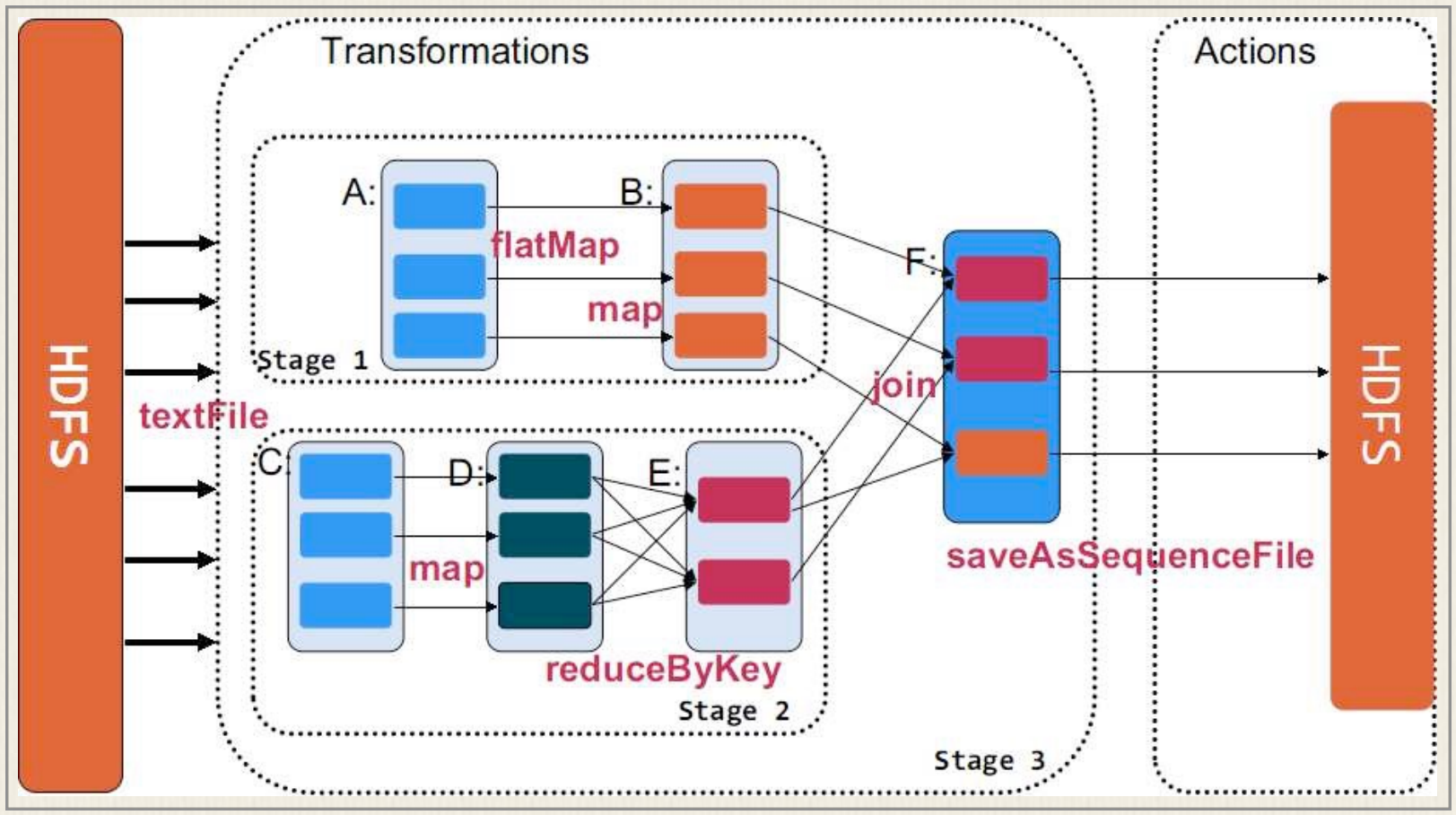


图1. Spark的核心机制图

在Spark的核心机制方面，主要有两个层面：首先是RDD（Resilient Distributed Datasets），RDD是Spark的最基本抽象，是对分布式内存的抽象使用，实现了以操作本地集合的方式来操作分布式数据集的抽象实现，它表示已被 分区，不可变的并能够被并行操作的数据集合，并且通常缓存到内存中，并且每次对RDD数据集的操作之后的结果，都可以存放到内存中，下一个操作可以直接从 内存中输入，省去了Map Reduce框架中由于Shuffle操作所引发的大量磁盘IO。这对于迭代运算比较常见的机器学习算法, 交互式数据挖掘来说，效率提升比较大。其次，就是在RDD上面执行的算子（Operator），在Spark的支持算子方面，主要有转换 （Transformation）和操作（Action）这两大类。在转换方面支持算子有 map， filter，groupBy和join等，而在操作方面支持算子有count，collect和save等。

Spark常见存储数据的格式是Key-Value，也就是Hadoop标准的Sequence File，但同时也听说支持类似Parquet这样的列存格式。Key-Value格式的优点在于灵活，上至数据挖掘算法，明细数据查询，下至复杂SQL 处理都能承载，缺点也很明显就是存储

空间比较浪费，和类似Parquet列存格式相比更是如此，key-Value格式数据一般是原始数据大小的2倍左 右，而列存一般是原始数据的1/3到1/4。

在效率层面，由于·使用Scala这样基于JVM的高级语言来构建，显而易见会有一定程度的损失，标准Java程序执行时候的速度基本接近C/C++ O0模式的程度，会比C/C++ O2模式的速度慢60%左右。

在技术创新方面，个人觉得Spark还谈不上创新，因为它其实属于比较典型In-Memory Data Grid内存数据网格，无论从7-8年前的IBM WebSphere eXtreme Scale到最近几年新出，并用于12306的Pivotal Gemfire都采用较类似的架构，都主要通过多台机器拼成一个较大内存网格，里面存储的数据都接近Key-Value模式，并且这个内存网格会根据很多 机制来确保数据会持久稳定地保存在内存中，并能保持数据的更新和恢复，而在网格上面使用一些常见的算子，来执行灵活的查询，并且用户可以写的程序来直接调 用这些算子。

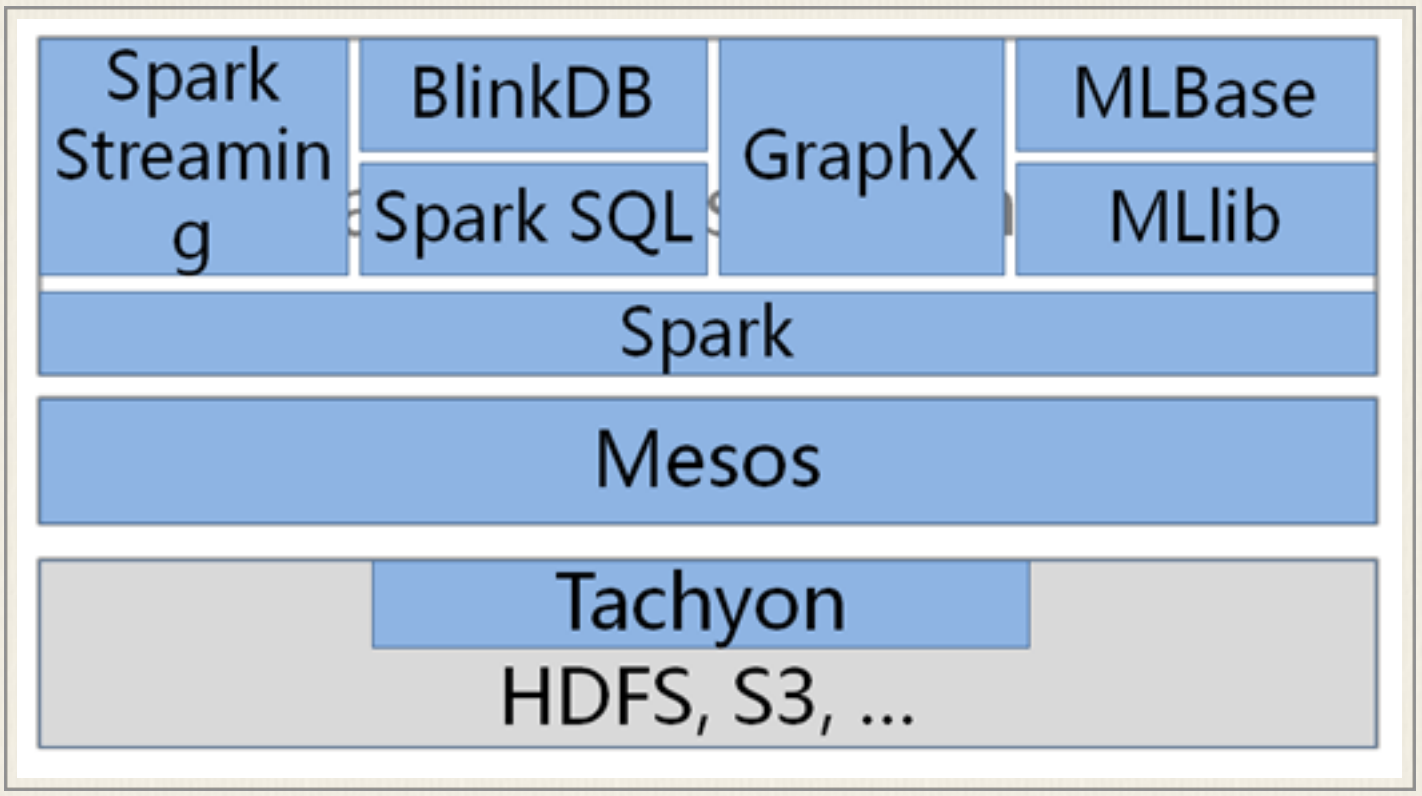


图2. Spark的生态圈

但是在整体架构的展现形式方面的，个人觉得Spark的确是领先同类开源产品两个身位的，因为它已经接近实现其Full Stack的梦想，它包括Spark Streaming，GraphX，MLBase，还有BlinkDB这个绝对的亮点（虽热个人觉得随着计算能力的提高，大数据在今后直接算也是可行 的）。还有，个

人真心对AMPLab的推广能力深深佩服。个人对Spark的总结是“创新的产品生态，较为传统的技术”。

SAP HANA的设计思路

其实至少10年前就有一波内存计算的风潮，那时代表性的产品主要有用于OLTP事务加速的Timeten和Altibase，而2010年开始的内 存计算技术产品，最有代表性的莫过于SAP HANA，由于HANA公开资料比较少，所以在技术方面的描述没办法像Spark那样的详细，那么我这边先根据部分公开的资料和我的一些理解稍微和大家聊 聊它使用到的一些核心技术。

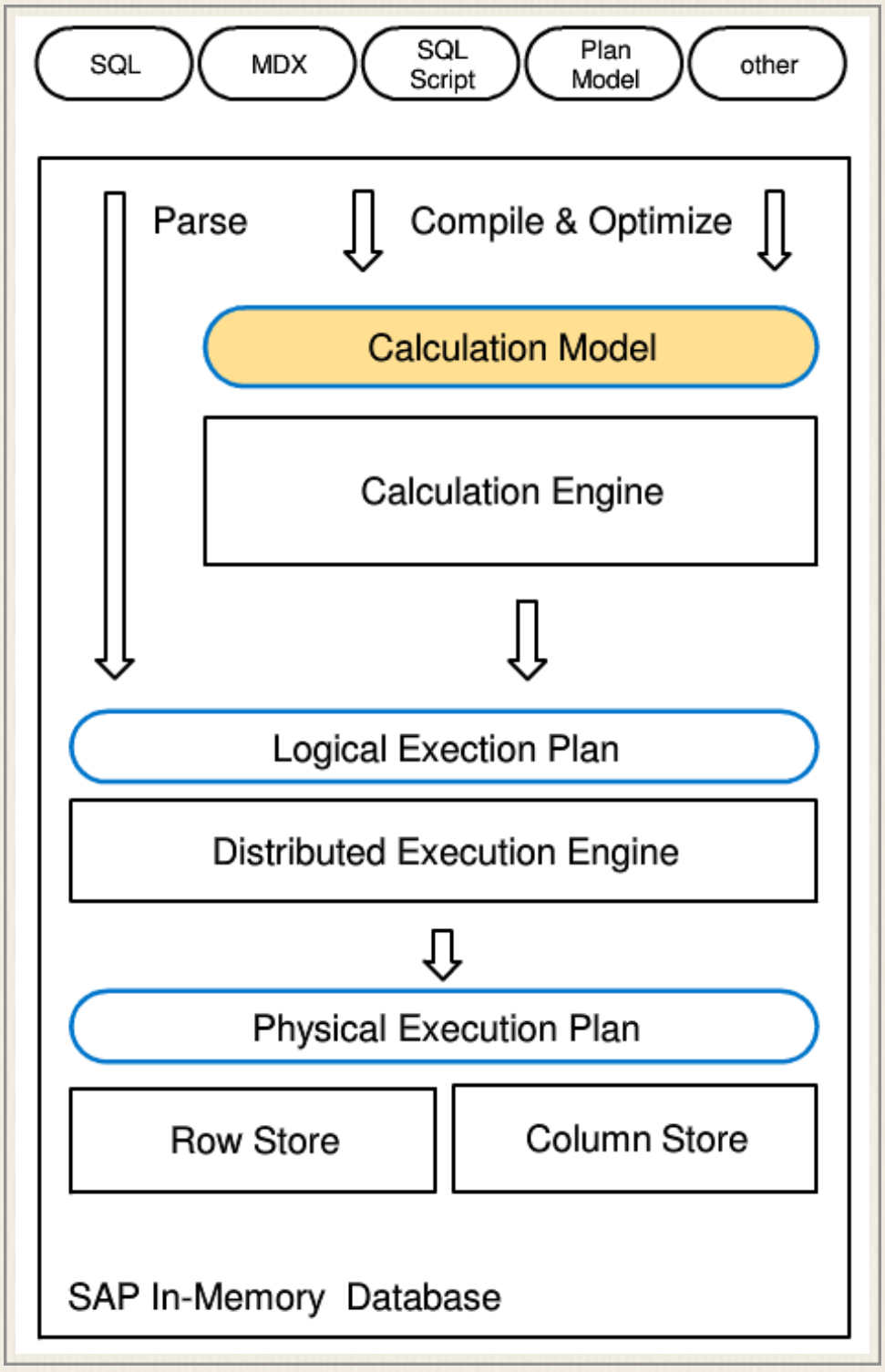


图3. SAP HANA计算引擎

主要有三个方面，首先，在性能优化方面，它尽可能地利用Intel x86 CPU特性，当然这是和他们在HANA设计初期就和德国Intel深度合作有关，主要做了两个设计：其一是全面利用最新的Intel指令集，在处理逻辑上面，全面采用Vector Processing的理念从而尽可能地使用最新的SSE4.1和SSE4.2等指令集，还有就是在NUMA场景下降低消耗，使其多线程性能提升参数尽可 能地接近1；其二是在数据结构方面，为了尽可能地利用好Cache，并尽可能少地访问内存，所以推出了缓存敏感的CSB（Cache Sensitive B+）树来代替传统的B树；其次，HANA还支持动态编译，无论是SQL查询还是MDX查询等，在HANA内部都会都被转译一个公共的表示层，名为L语 言，并且在执行之前会使用LLVM来进行编译为二进制代码，并执行，这样做的好处主要是避免传统数据库引擎繁琐的Switch-Case逻辑，并且由于这 些Switch-Case逻辑很容易导致Context切换，所以如果避免类似的逻辑，这样对整体性能裨益良多；还有就是完全内存化，也就是确保所有数据 都在内存中，就算是用来做数据安全性的Snapshot快照也不使用廉价的硬盘，而是使用昂贵的SSD来做保存，这样保存和恢复都更快。

在存储数据结构方面，HANA是行存和列存都支持，但是根据我碰到的一些用户反馈，用户基本上还是以使用列存为主。

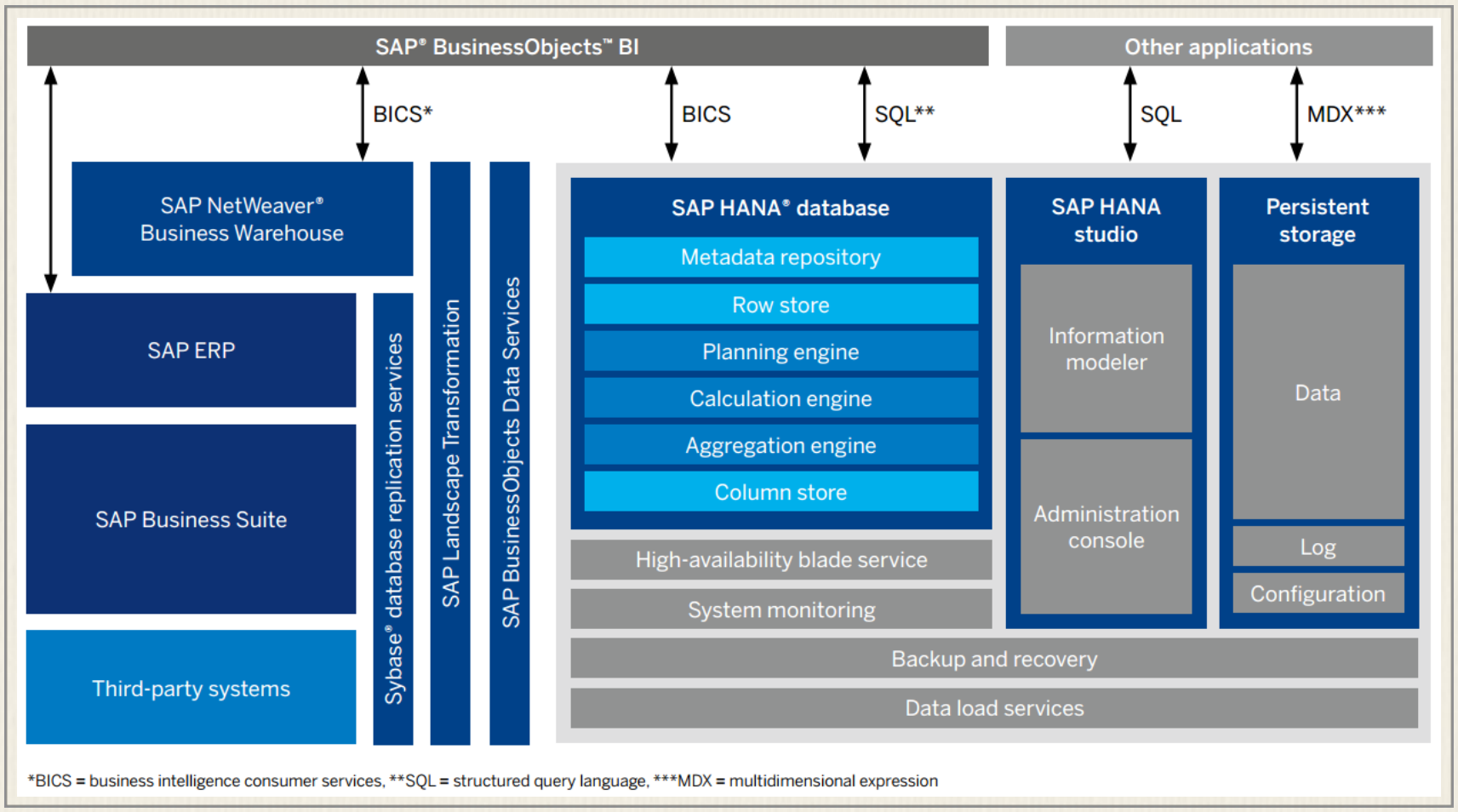


图4. SAP HANA产品全貌

在产品形态方面，它主要还是提供多种工具和产品接入，都主要以分析为主，比如类似SAP NetWeaver或者BO这样BI工具，还有支持文本分析，以及各种预测算法，并且在这些之上，开发出很多针对某些行业的应用，比如，财务方面，物流方 面和广告方面的，所以根据部分用户的反馈，HANA如果只是当它内存数据库来用，其实价值不是特别大，但是如果能把它当中开发平台来使用，那么就很物尽其用，因为它上面能利用的库和应用比较多。在 销售方式方面，还是传统的License 模式。总体而言，个人觉得SAP HANA 这样内存计算平台“有特色的技术，较传统的产品形态”。

综述

为什么要聊聊内存计算这个问题，因为我基于个人多年的研发经验，对于常见的SQL分析而言，由于其本身读写形式是连续读，而连续读硬盘本身的读写能 力也是挺强的，再加上存储数据本身是压缩的，所以当硬盘个数和CPU个数比较匹配的话（比如1：1），那么在执行数据分析的时候，数据是否在内存并不是极 为关键，性能比在1比6左右，也就是数据完全在内存比数据完全在硬盘中快5倍左右，这个性能比在大多数情况下用户不会觉得非常关键，所以个人觉得单纯把全 部数据放在内存中的意义不是特别大，因此我特地拿出Apache Spark和SAP HANA这两款产品的出来比较，从而发觉现在其实内存计算没那么简单，还是有非常多的门道的。那么对于用户，该如何在这两种技术之间选择呢？下面是我个人 的见解：

对于那些希望有一整套Full Stack的支持初创企业，个人支持你们去使用Spark，因为他们这个群体本身的特色就是喜欢尝试新鲜的东西，数据不会特别大，需求会比较多变，同时也不会使用到特别复杂的功能，所以Spark对他们而言，更适合。

对于HANA的，个人觉得特别适合那些传统企业，因为它的SQL接口更成熟，速度更快，可以做到复杂查询实时出结果，于此同时它提供的文本分析工具 和数据挖掘工具，但可惜许可证成本太高，并且也因为这个原因，导致使用HANA的群体比较小，没有一个生态群，所以HANA技术上的创新也很难造福千千万 万的程序员。

吴朱华：上海云人信息科技有限公司的联合创始人兼CEO，国内资深的云计算和大数据专家，之前曾在IBM中国研究院参与过多款云计算产品的开发 工作，同济本科，并曾在北京大学读过硕士。2010年底，他和另两位创始人组建了一支十多人的团队，在上海杨浦云基地办公。云人信息科技有限公司目前专注 于大数据实时分析，尤其是互联网广告、运营商、证券金融和智能电网等有大数据实时分析需求的行业与企业。2011年中，发表业界最好的两本云计算书之一 《云计算核心技术剖析》。在2013年以唯一云计算和大数据的代表初入选“2013年福布斯中国30位30岁以下的创业者”。

原文链接：http://www.valleytalk.org/2014/11/11/%E5%86%85%E5%AD%98%E8%AE%A1%E7%AE%97%E6%8A%80%E6%9C%AF%E9%82%A3%E5%AE%B6%E5%BC%BA%EF%BC%9Fspark-vs-hana/

编译器的工作过程

作者：阮一峰

源码要运行，必须先转成二进制的机器码。这是编译器的任务。

比如，下面这段源码（假定文件名叫做test.c）。

```
#include <stdio.h>

int main(void)
{
  fputs("Hello, world!\n", stdout);
  return 0;
}
```

要先用编译器处理一下，才能运行。

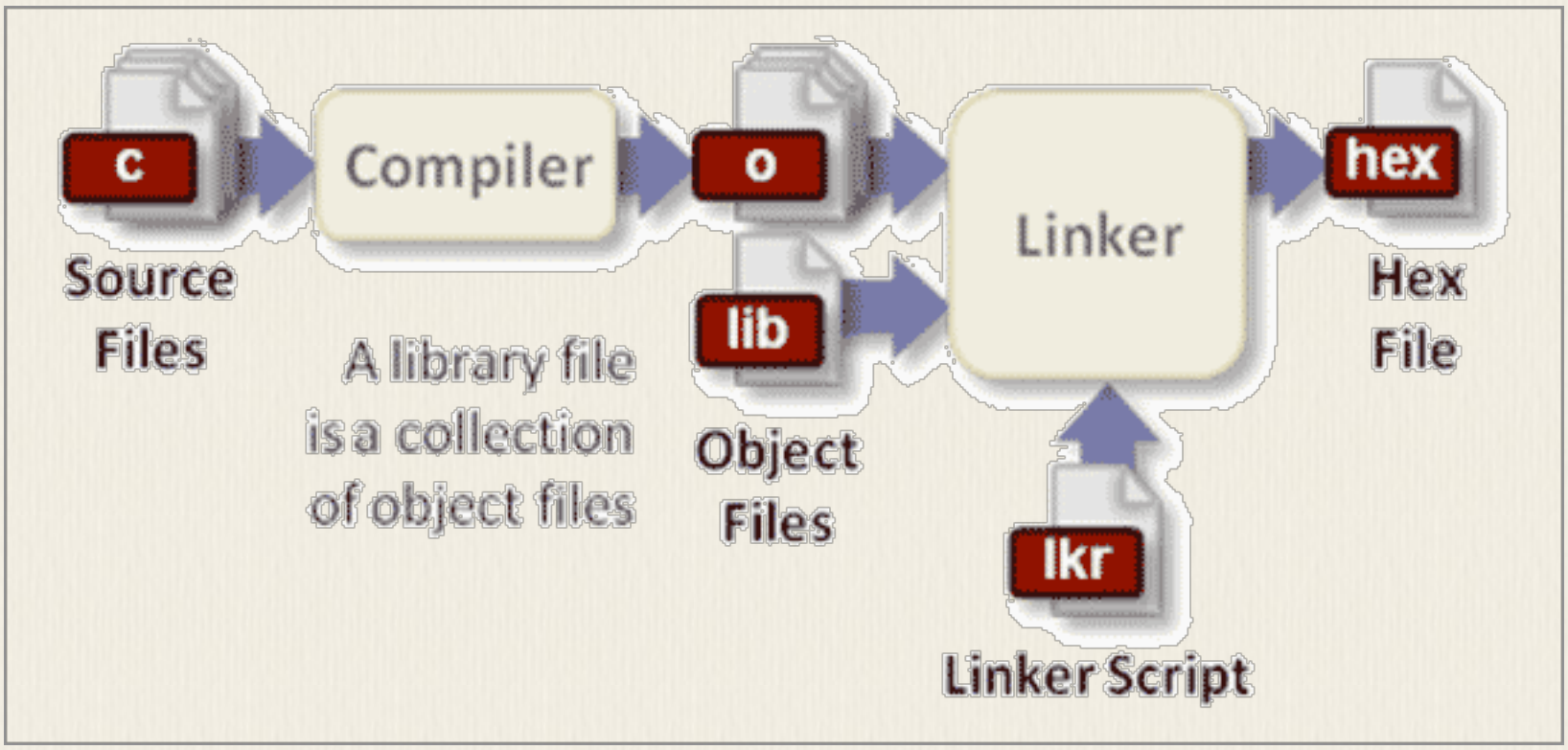
```
$ gcc test.c
$ ./a.out
Hello, world!
```

对于复杂的项目，编译过程还必须分成三步。

```
$ ./configure
$ make
$ make install
```

这些命令到底在干什么？大多数的书籍和资料，都语焉不详，只说这样就可以编译了，没有进一步的解释。

本文将介绍编译器的工作过程，也就是上面这三个命令各自的任务。我主要参考了Alex Smith的文章《Building C Projects》。需要声明的是，本文主要针对gcc编译器，也就是针对C和C++，不一定适用于其他语言的编译。



第一步 配置（configure）

编译器在开始工作之前，需要知道当前的系统环境，比如标准库在哪里、软件的安装位置在哪里、需要安装哪些组件等等。这是因为不同计算机的系统环境不 一样，通过指定编译参数，编译器就可以灵活适应环境，编译出各种环境都能运行的机器码。这个确定编译参数的步骤，就叫做"配置"（configure）。

这些配置信息保存在一个配置文件之中，约定俗成是一个叫做configure的脚本文件。通常它是由autoconf工具生成的。编译器通过运行这个脚本，获知编译参数。

configure脚本已经尽量考虑到不同系统的差异，并且对各种编译参数给出了默认值。如果用户的系统环境比较特别，或者有一些特定的需求，就需要手动向configure脚本提供编译参数。

$ ./configure --prefix=/www --with-mysql

上面代码是php源码的一种编译配置，用户指定安装后的文件保存在www目录，并且编译时加入mysql模块的支持。

第二步 确定标准库和头文件的位置

源码肯定会用到标准库函数（standard library）和头文件（header）。它们可以存放在系统的任意目录中，编译器实际上没办法自动检测它们的位置，只有通过配置文件才能知道。

编译的第二步，就是从配置文件中知道标准库和头文件的位置。一般来说，配置文件会给出一个清单，列出几个具体的目录。等到编译时，编译器就按顺序到这几个目录中，寻找目标。

第三步 确定依赖关系

对于大型项目来说，源码文件之间往往存在依赖关系，编译器需要确定编译的先后顺序。假定A文件依赖于B文件，编译器应该保证做到下面两点。

（1）只有在B文件编译完成后，才开始编译A文件。

（2）当B文件发生变化时，A文件会被重新编译。

编译顺序保存在一个叫做makefile的文件中，里面列出哪个文件先编译，哪个文件后编译。而makefile文件由configure脚本运行生成，这就是为什么编译时configure必须首先运行的原因。

在确定依赖关系的同时，编译器也确定了，编译时会用到哪些头文件。

第四步 头文件的预编译（precompilation）

不同的源码文件，可能引用同一个头文件（比如stdio.h）。编译的时候，头文件也必须一起编译。为了节省时间，编译器会在编译源码之前，先编译头文件。这保证了头文件只需编译一次，不必每次用到的时候，都重新编译了。

不过，并不是头文件的所有内容，都会被预编译。用来声明宏的#define命令，就不会被预编译。

第五步 预处理（Preprocessing）

预编译完成后，编译器就开始替换掉源码中bash的头文件和宏。以本文开头的那段源码为例，它包含头文件stdio.h，替换后的样子如下。

```
extern int fputs(const char *, FILE *);
extern FILE *stdout;

int main(void)
{
    fputs("Hello, world!\n", stdout);
    return 0;
}
```

为了便于阅读，上面代码只截取了头文件中与源码相关的那部分，即fputs和FILE的声明，省略了stdio.h的其他部分（因为它们非常长）。另外，上面代码的头文件没有经过预编译，而实际上，插入源码的是预编译后的结果。编译器在这一步还会移除注释。

这一步称为"预处理"（Preprocessing ），因为完成之后，就要开始真正的处理了。

第六步 编译（Compilation）

预处理之后，编译器就开始生成机器码。对于某些编译器来说，还存在一个中间步骤，会先把源码转为汇编码（assembly），然后再把汇编码转为机器码。

下面是本文开头的那段源码转成的汇编码。

```
    .file   "test.c"
    .section    .rodata
.LC0:
    .string "Hello, world!\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
```

```
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    stdout(%rip), %rax
    movq    %rax, %rcx
    movl    $14, %edx
    movl    $1, %esi
    movl    $.LC0, %edi
    call    fwrite
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Debian 4.9.1-19) 4.9.1"
    .section    .note.GNU-stack,"",@progbits
```

这种转码后的文件称为对象文件（object file）。

第七步 连接（Linking）

对象文件还不能运行，必须进一步转成可执行文件。如果你仔细看上一步的转码结果，会发现其中引用了stdout函数和fwrite函数。也就是说，程序要正常运行，除了上面的代码以外，还必须有stdout和fwrite这两个函数的代码，它们是由C语言的标准库提供的。

编译器的下一步工作，就是把外部函数的代码（通常是后缀名为.lib和.a的文件），添加到可执行文件中。这就叫做连接（linking）。这种通过拷贝，将外部函数库添加到可执行文件的方式，叫做静态连接（static linking），后文会提到还有动态连接（dynamic linking）。

make命令的作用，就是从第四步头文件预编译开始，一直到做完这一步。

第八步 安装（Installation）

上一步的连接是在内存中进行的，即编译器在内存中生成了可执行文件。下一步，必须将可执行文件保存到用户事先指定的安装目录。

表面上，这一步很简单，就是将可执行文件（连带相关的数据文件）拷贝过去就行了。但是实际上，这一步还必须完成创建目录、保存文件、设置权限等步骤。这整个的保存过程就称为"安装"（Installation）。

第九步 操作系统连接

可执行文件安装后，必须以某种方式通知操作系统，让其知道可以使用这个程序了。比如，我们安装了一个文本阅读程序，往往希望双击txt文件，该程序就会自动运行。

这就要求在操作系统中，登记这个程序的元数据：文件名、文件描述、关联后缀名等等。Linux系统中，这些信息通常保存在/usr/share/applications目录下的.desktop文件中。另外，在Windows操作系统中，还需要在Start启动菜单中，建立一个快捷方式。

这些事情就叫做"操作系统连接"。make install 命令，就用来完成"安装"和"操作系统连接"这两步。

第十步 生成安装包

写到这里，源码编译的整个过程就基本完成了。但是只有很少一部分用户，愿意耐着性子，从头到尾做一遍这个过程。事实上，如果你只有源码可以交给用 户，他们会认定你是一个不友好的家伙。大部分用户要的是一个二进制的可执行程序，立刻就能运行。这就要求开发者，将上一步生成的可执行文件，做成可以分发 的安装包。

所以，编译器还必须有生成安装包的功能。通常是将可执行文件（连带相关的数据文件），以某种目录结构，保存成压缩文件包，交给用户。

第十一步 动态连接（Dynamic linking）

正常情况下，到这一步，程序已经可以运行了。至于运行期间（runtime）发生的事情，与编译器一概无关。但是，开发者可以在编译阶段选择可执行 文件连接外部函数库的方式，到底是静态连接（编译时连接），还是动态连接（运行时连接）。所以，最后还要提一下，什么叫做动态连接。

前面已经说过，静态连接就是把外部函数库，拷贝到可执行文件中。这样做的好处是，适用范围比较广，不用担心用户机器缺少某个库文件；缺点是安装包会 比较大，而且多个应用程序之间，无法共享库文件。动态连接的做法正好相反，外部函数库不进入安装包，只在运行时动态引用。好处是安装包会比较小，多个应用 程序可以共享库文件；缺点是用户必须事先安装好库文件，而且版本和安装位置都必须符合要求，否则就不能正常运行。

现实中，大部分软件采用动态连接，共享库文件。这种动态共享的库文件，Linux平台是后缀名为.so的文件，Windows平台是.dll文件，Mac平台是.dylib文件。

（文章完）

原文链接：http://www.ruanyifeng.com/blog/2014/11/compiler.html

如何给你的Android 安装文件（APK）瘦身

作者：Cyril Mottier

Android的apk文件越来越大了这已经是一个不争的事实。在Android 还是最初版本的时候，一个app的apk文件大小也还只有2 MB左右，到了现在，一个app的apk文件大小已经升级到10MB到20MB这个范围了。apk文件大小的爆炸式增长主要是因为用户对app质量的期待 越来越高以及开发者的开发经验增长，具体体现在以下几个方面：

- Android设备 dpi 的多样化 ([l|m|tv|h|x|xx|xxx]dpi)
- Android 平台的进化，开发工具的改进以及开源类库生态系统的丰富
- 用户对高质量UI的期待
- 其他原因

Android开发者在设计一个app的时候应该将最终发布一个轻量级app作为一个最佳实践来考虑。为什么？首先这就意味着你拥有了一个简洁，易 维护代码基础。其次，官方应用商店对超过50MB的apk设置了拓展包文件下载选项，apk文件在50MB以下更容易让用户下载。最后，我们的应用程序环 境是一个带宽有限，存储空间有限的环境，apk安装文件越小，下载就会越快，安装也会更快，良性循环，最后说不定用户因为这个给好评。

在大部分情况下，apk大小的增长是为了满足消费者的需要和期待。然而，我认为apk大小的增速已经超过了用户对app期待的增速。所以，很大程度 上，官方应用商店里面的那些程序可以瘦身至它们现在大小的一半甚至更多。在这篇文章里面，我将写下一些关于如何给apk文件瘦身的招式，希望你们能够喜欢。

APK 文件格式

在说如何给apk瘦身之前，让我们先来看看apk文件内部的结构到底是怎么一回事。说简单点，一个apk文件就是包含一些文件的压缩包。作为开发者，我们通过使用 unzip 命令解压这个apk文件一探apk的内部结构。下面的文件结构就是我们使用 unzip <your_apk_name>.apk1这个命令所获得的：

/assets

/lib

/armeabi

/armeabi-v7a

/x86

/mips

/META-INF

MANIFEST.MF

CERT.RSA

CERT.SF

/res

AndroidManifest.xml

classes.dex

resources.arsc

我们可能对上面大部分的文件和目录都很熟悉。它们和我们在实际开发app的时候所看到得项目结构一样，包含了： /assets, /lib, /res, AndroidManifest.xml. 还有一些文件可能是我们第一次看到。一般说来，classes.dex, 它包含了我们所写的Java代码经过编译后class文件；

resources.arsc 包含了预编译之后的资源文件（比如values文件，XML drawables 文件等。）。

由于apk文件只是一个简单地压缩文件，这就意味着它有两种大小：即压缩前的大小和压缩后的大小。这篇文章我将主要关注压缩后的大小。

如何减少apk文件大小

减少apk文件大小可以从几个方面入手。由于每个app都是不同的，所以没有什么绝对规则来给apk文件瘦身。作为apk文件的三个重要组成部分，我们可以考虑从它们开始入手：

- Java 源代码
- 资源文件（resources/assets）
- native code

所以接下来的招式都是由减少这些组件的大小出发，进而减少整个app的大小。

掌握良好的编码习惯

这是减少apk文件至关重要的第一步。你要对自己的代码了如子掌。你要移除掉所有无用处的dependency libraries，让你的代码一天比一天优秀，持续地优化你的代码。总而言之，保持一个简洁，最新的代码基础是减少apk文件至关重要的一环。

当然，从零开始一个项目并为这个项目保持一份简洁的代码基础很容易。项目越老，这个工作就越困难。事实上，拥有一大段历史背景的项目必须要去处理各种死代码和无用代码。还好有许多的开发工具可以帮我们来做这些事情……

使用 Proguard

Proguard 是一个很强悍的工具，它可以帮你在代码编译时对代码进行混淆，优化和压缩。它有一个专门用来减少apk文件大小的功能叫做 tree-shaking。Proguard 会遍历你的所有代码然后找出无用处的代码。所有这些不可达（或者不需要）的代码都会在生成最终的apk文件之前被清除掉。Proguard 也会重命名你的类属性，类和接口，然整个代码尽可能地保持轻量级水平。

也许现在你会认为 Proguard 是一个相当有效地工具。但是能力越大，责任也就越大。现在许多开发这认为Proguard有点让人不省心，因为它会重度依赖反射。哪些类或者属性需要被处理或者不能处理都要开发者对 Proguard 进行配置。

广泛使用 Lint

Proguard 只会对 Java 代码起作用，那么对哪些资源文件呢？比如一张图片 my_image 在 res/drawable 文件夹中，没有被使用，Proguard 只会移除掉 R 类中的引用，但是图片依然还在文件夹中。

Lint 一个静态的代码分析器，你只需通过调用 ./gradlew lint这个简单地命令它就能帮你检查所有无用 的资源文件。它在检测完之后会提供一份详细的资源文件清单，并将无用的资源列在“UnusedResources: Unused resources” 区域之下。只要你不通过反射来反问这些无用资源，你就可以放心地移除这些文件了。

Lint 会分析资源文件(比如 /res 文件夹下面的文件) ，但是会跳过 assets 文件 (/assets 文件夹下面的文件)。事实上assets 文件是可以通过它们的文件名直接访问的，而不需要通过Java引用或者XML引用。因此，Lint 也不能判定某个 asset 文件在项目中是否有用。这全取决于开发者对这个文件夹的维护了。如果你没有使用某个asset 文件，那么你就可以直接清除这个文件。

对资源文件进行取舍

Android 支持多种设备。Android的系统设计让它可以支持设备的多样性：屏幕密度，屏幕形状，屏幕大小等等。到了Android 4.4，它支持的屏

幕密度包括： ldpi, mdpi, tvdpi, hdpi, xhdpi, xxhdpi and xxxhdpi。但是你要知道的一点是，Android 支持这么多的屏幕密度并不意味着你需要为每一个屏幕密度提供相应的资源文件。

如果你知道某些屏幕密度的设备只有很少部分用户在使用，那么你就可以直接不需要使用相应屏幕密度的资源文件。就我个人而言，我只会为我的应用提供 hdpi, xhdpi and xxhdpi2 这几个屏幕密度的支持。如果某些设备不是这几个屏幕密度的，不用担心，Android 系统会自动使用存在的资源为设备计算然后提供资源文件。

我这么做得原因很简单。首先，这些设备屏幕密度就能覆盖我 80% 的用户。其次，xxxhdpi 这个屏幕密度只是在为未来的设备做准备，但是未来还未到来。最后，我真的不怎么关心低屏幕密度（比如mdpi 和 ldpi），无论我为这几个屏幕密度努力，结果都是令人伤心地，还不如直接让Android系统对 hdpi 资源文件进行适当地缩放来匹配相应地低端机型。

同样地，在 drawable-nodpi 文件夹里面维持一个文件也能节省空间。当然前提是你觉得对这个文件进行相应地缩放之后呈现的效果你能接受或者这个文件出现的几率很少。

资源文件最少化配置

Android 开发经常会依赖各种外部开源代码库，比如Android Support Library, Google Play Services, Facebook SDK 等等。但是这些库里面并不是所有的资源文件你都会用到。比如， Google Play Services 里面会有一些为其他语种提供翻译，而你的app又不需要这个语种的翻译，而且这个库里面还包含了我的app中不支持的 mdpi 资源文件

还好从Android Gradle Plugin 0.7 开始，你可以配置你的app的build系统。这主要是通过配置 resConfig 和 resConfigs 以及默认的配置选项。下面的 DSL （Domain Specific Language）就会阻止 aapt（Android Asset Packaging Tool）打包app中不需要的资源文件。

defaultConfig {

```
    // ...

    resConfigs "en", "de", "fr", "it"
    resConfigs "nodpi", "hdpi", "xhdpi", "xxhdpi", "xxxhdpi"
}
```

压缩图片

Aapt（Android Asset Packaging Tool）就内置了 保真图像压缩算法。例如，一个只需 256 色的真彩PNG图片会被aapt 通过一个颜色调色板转化成一个 8-bit PNG 文件。这可以帮助你减少图片文件的大小。当然你还可以通过Google查找相应的优化工具，比如 pngquant, ImageAlpha 和 ImageOptim 等。你可以从中选择一个适合你的工具。

还有一种只在Android平台上存在的图片文件也可以优化，它就是9-patches。就我目前所知道，我还没发现有这个文件的优化工具。然而你只需要求你的设计师将它的可扩展区域和内容区域尽可能地减少即可。这不但可 以减少资源文件的大小，还能使得以后资源文件的维护变得更加简单。

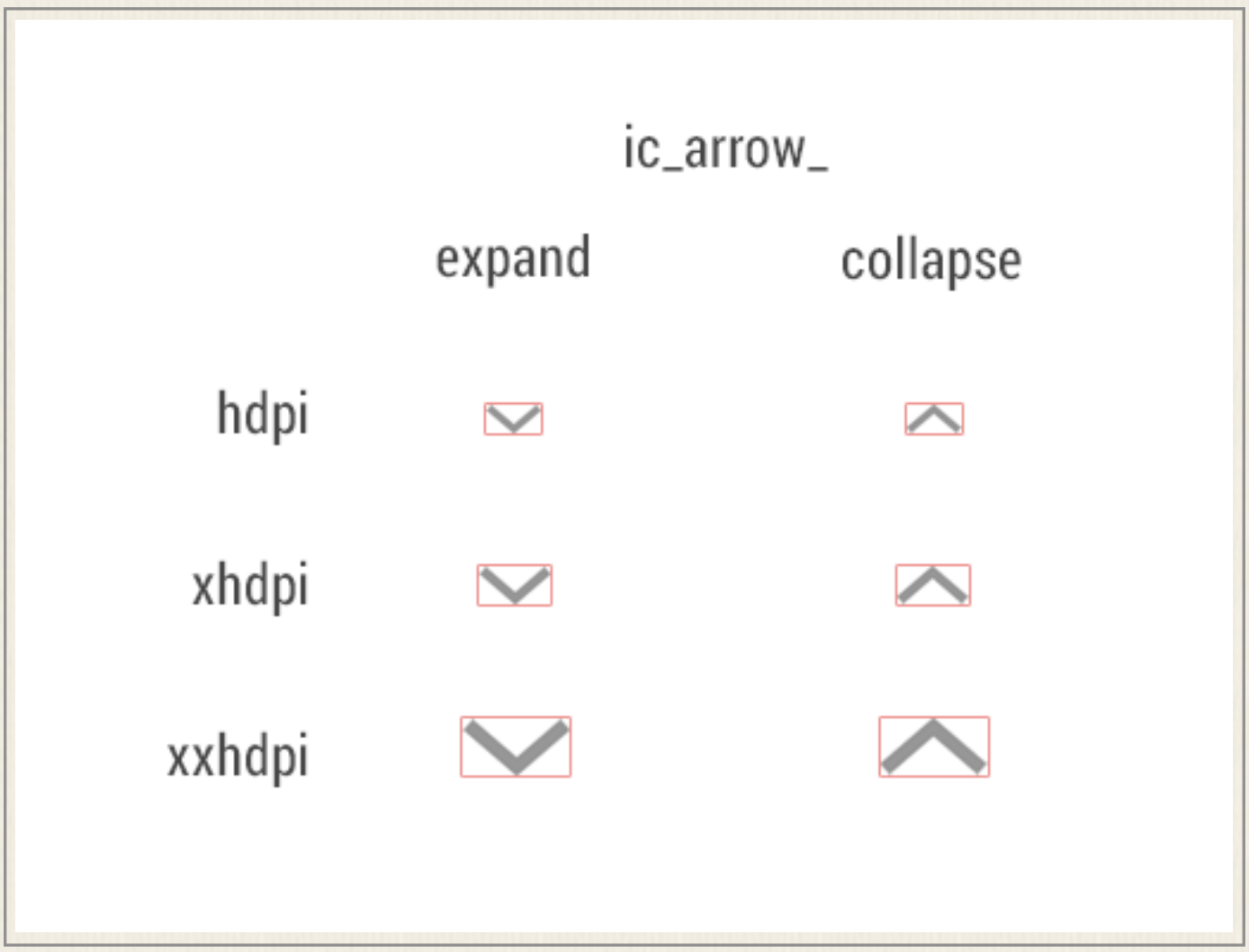
限制app支持的cpu 架构的数目

一般说来Android 使用Java 代码即可以满足大部分需求，不过还是有一小部分案例需要使用一些 native code。就像之前对资源文件那样opinionated，你可以这些 native code opinionated。 在当前的Android 生态系统中，让你的app支持 armabi 和 x86 架构就够了。这里有一篇相当不错的关于如何瘦身native 代码库的文章，你可以参考参考。

尽可能地重用

重用资源可能是你在进行移动开发时需要了解的最重要的优化技巧之一。比如在一个 ListView 或者 RecyclerView，重用可以帮助你在列表滚动时保持界面流畅。重用还可以帮你减少apk文件的大小。例如，Android 提供了几个工具为一个asset文件重新着色，在Android L中你可以使用 android:tint 和 android:tintMode 来达到效果，在老版本中则可以使用 ColorFilter 。

如果系统中有两种图片，一种图片是另一种图片翻转180°得到的，那么你就可以移除一种图片，通过代码实现。比如你现在有两种图片分别命名为 ic_arrow_expand 和 ic_arrow_collapse :



你可以直接移除掉 ic_arrow_collapse 文件，然后在ic_arrow_expand 的基础上创建一个 RotateDrawable 。这个方法也可以让你减少设计人员的工作:

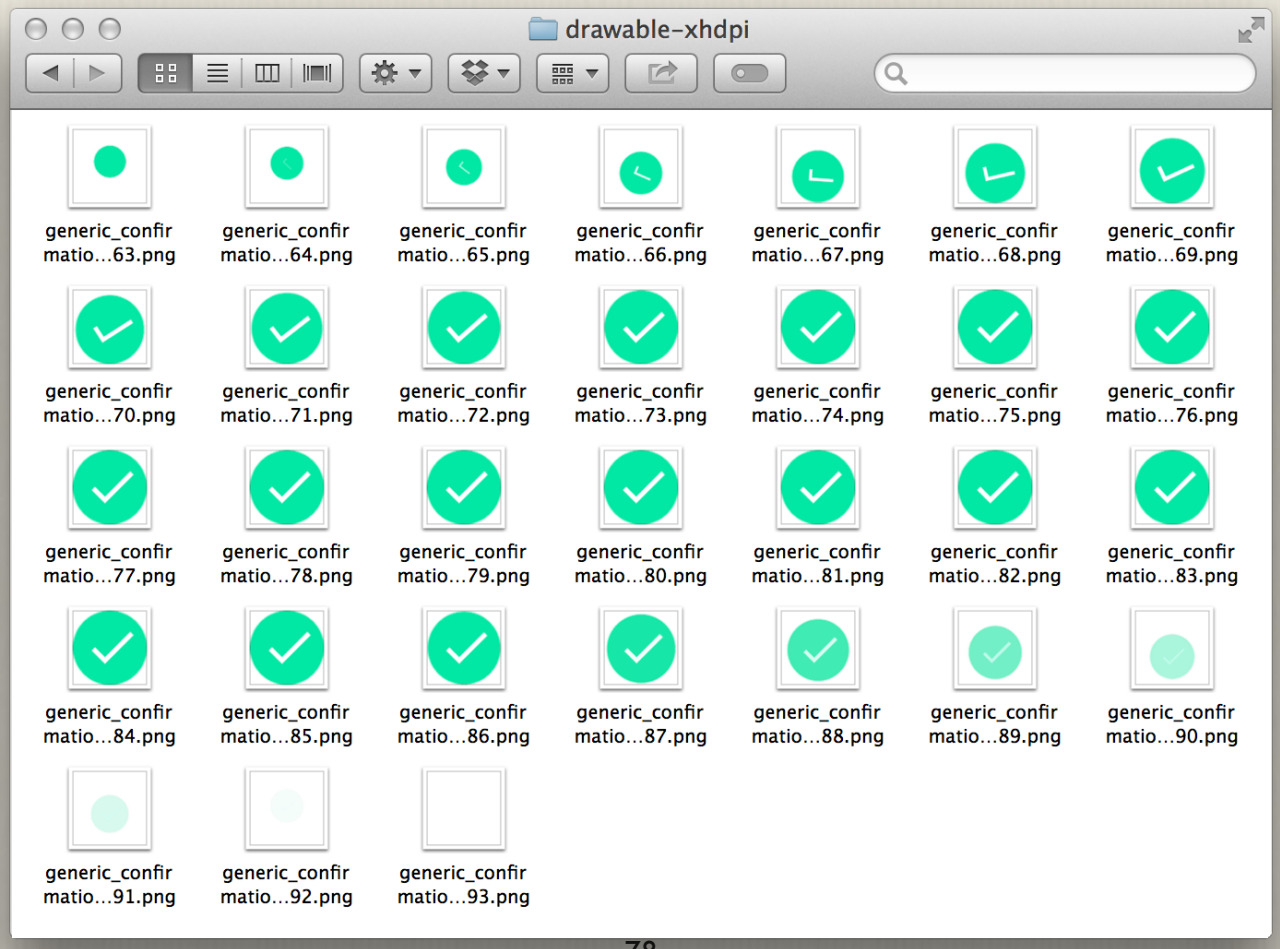
```
<?xml version="1.0" encoding="utf-8"?>

<rotate xmlns:android="http://schemas.android.com/apk/res/android"
```

```
android:drawable="@drawable/ic_arrow_expand"
android:fromDegrees="180"
android:pivotX="50%"
android:pivotY="50%"
android:toDegrees="180" />
```

在合适的时候使用代码渲染图像

在某些情况下，直接使用Java 代码渲染图像也能获得不错的效果。比如逐帧动画就是一个很好的例子。最近我都在尝试一些Android Wear 的开发，了解了一下Android wearable support library。就像其他的Android support library 一样，这个库里面也有一些工具来处理穿戴设备的。

不过让我吃惊的是，当我简单地构建了一个 “Hello World”示例，最后得到的apk文件竟然有1.5MB。于是我快速地研究了一下 wearable-support.aar 文件，发现这个库有两个逐帧动画，并分别支持了3种不同的屏幕密度：一个 “success” 动画 (31 frames) 和一个 “open on phone” 动画 (54 frames)。



这个逐帧success动画是被一个叫做 AnimationDrawable 所定义的：

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
xmlns:android="http://schemas.android.com/apk/res/android"
android:oneshot="true">
    <item android:drawable="@drawable/generic_confirmation_00163"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00164"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00165"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00166"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00167"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00168"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00169"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00170"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00171"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00172"
android:duration="33"/>
    <item android:drawable="@drawable/generic_confirmation_00173"
android:duration="33"/>
```

```
        <item android:drawable="@drawable/generic_confirmation_00174"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00175"
android:duration="333"/>

        <item android:drawable="@drawable/generic_confirmation_00185"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00186"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00187"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00188"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00189"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00190"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00191"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00192"
android:duration="33"/>

        <item android:drawable="@drawable/generic_confirmation_00193"
android:duration="33"/>

    </animation-list>
```

这样做得好处就是 (我当然在讽刺) 每帧显示33ms，这使得整个动画保持在30fps的频率。如果每帧16ms这将会导致整个库是之前的两倍大。如果你去看源码你会发现很有趣。在 generic_confirmation_00175 这一帧 (15 行) 将持续显示 333ms。 generic_confirmation_00185 紧跟着它。这个优化节省了9个类似的帧 (包含了从176 帧到 184 帧) 。不过最后神奇的是 wearable-support.aar 竟然神奇的包含了这个9个完全无用的帧，而且还以3中屏幕密度展示。3

在代码中来渲染这样的动画明显会很花时间。然而当你维持动画运行在60fps这样的频率可以大幅度的减少apk的大小。在写这篇博文的时 候，Android还没提供工具来渲染这样的动画。但是我希望Google正在开发新一代的轻量级实时渲染系统来保证material design的细节呈现。当然“Adobe After Effect to VectorDrawable” 之类的设计工具也能提供很多方便。

如何更进一步?

上面所有的招式都集中在app或者library 开发者。也许我们还可以在app分发渠道方面为apk大小做出一些改变？我想可以在app 分发服务器端做一些改进，或者在官方应用商店。例如，我们可以期待官方应用商店在用户安装app的时候为设备绑定相应的native 库而摒弃那些无用的。

同样地，我们还可以想象只根据目标设备的配置来打包应用。不幸的是，这可能破坏Android 生态一个重要的功能特性：配置热置换。事实上，Android一开始就是位处理各种实时的配置更改（语言，屏幕转向）而设计的。如果我们移除掉与目标屏幕 不兼容的资源文件，这可以极大的减少文件大小。不过Android需要处理实时的屏幕密度更改。即便我们假设废除这种功能，我们仍然需要处理为不同的屏幕密度设计的图片以及其他配置（比如屏幕朝向，最小宽度等）。

服务器端的apk打包看起来很强大。但这样会冒很大得风险，因为最终传送给用户的apk会于开发者发给的服务器的完全不同。分发一些缺失资源文件的apk可能会导致app崩溃。

总结

设计就是在一个约束集里面找出最好的方案。显然apk文件的大小就是一个约束。不要害怕为了让多个方面变得更好而放松一个方面的约束。例如，当你要 降低UI的渲染效果时，不要犹豫，因为这可以让apk的大小减小，同时使得app的运行也更加流畅。你99%的用户是感受不到UI质量变低的，但是他们会 注意到apk文件变小了，运行也更加流畅了。总之，你需要将app各方面进行整体考虑，而不是仅仅几个方面的斟酌。

原译文链接：http://greenrobot.me/devpost/putting-your-apks-on-diet/

原文链接：http://cyrilmottier.com/2014/08/26/putting-your-apks-on-diet/

小米11.11：海量数据压力下的推送服务

作者：臧秀涛

11.11大促，随着移动端业务量的急剧提升，像小米推送这样的基础服务也经受了巨大的考验。11月12日，小米的项目总监汪轩然在微博上宣布，“小米推送服务共发出9.65亿条消息，平均每分钟发送67万条。更值得一提的是，后台监控显示，推送服务后台系统在全天运作非常平稳，没有任何卡顿拥堵现象，让各种促销、返利、订单更新消息第一时间触达用户。”

汪轩然，2007年毕业于清华大学计算机系，后加入微软亚洲工程院，曾参与WP7上的浏览器的开发。2010年7月加入 小米，曾担任米聊安卓团队的团队主管，现在在小米任项目总监，负责小米的开发者服务，掌管推送服务、统计服务和移动广告联盟三大业务，旨在为小米搭建一个 移动App业务的互联网生态圈。

我们联系了汪轩然，就小米推送服务的架构、特点、性能等问题对他进行了采访，以下内容根据本次采访整理而成。

基础技术架构

协议是推送服务的核心。小米推送服务所采用的协议是由之前的米聊演变过来的，而米聊 从一开始就选择使用XMPP协议，之后开发团队对XMPP协议做过几轮精简和重构。现在XMPP部分只是作为一个数据的传输层，之上跑着各种独立的业务， 每个业务称为一个“channel”；每个channel上跑的数据格式可以是不一样的。消息推送服务是其中一个channel，这个channel上传 输的数据是通过Thrift进行二进制化的协议格式。

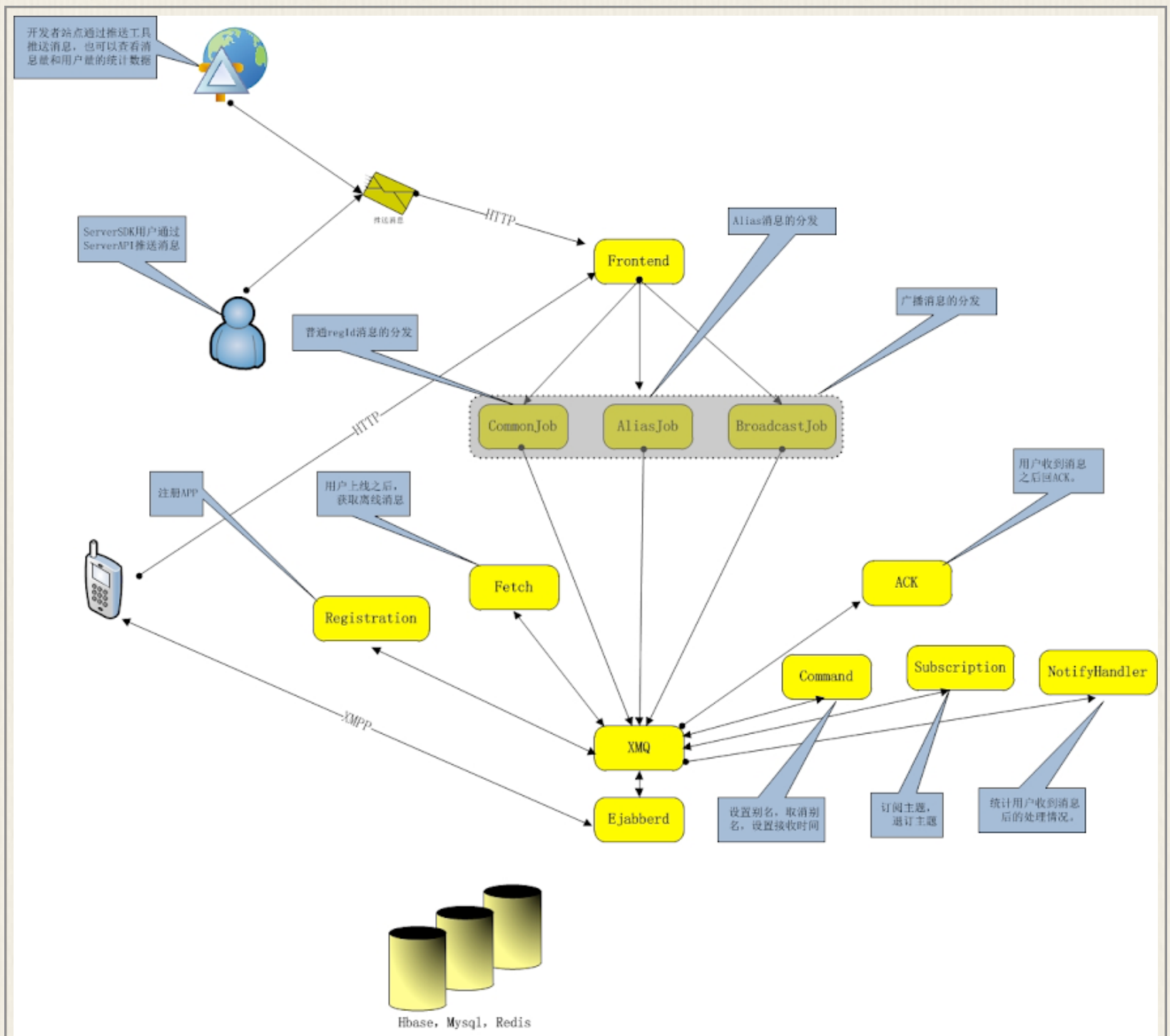
再来看一下小米推送服务的服务端架构。下图是后台服务端的一个基本架构图。整个服务端包含如下几层：

1. XMPP前端：用于维护跟客户端之间的长连接，使用EJabberd项目来处理来自客户端的XMPP请求，同时通过XMQ模块来处理推送服务特有的XMPP消息协议。

2. 中间层：业务逻辑层，主要用于将消息请求异步化、创建和维护消息队列、以及处理客户端的一些命令请求（注册、设置别名、设置topic等）。

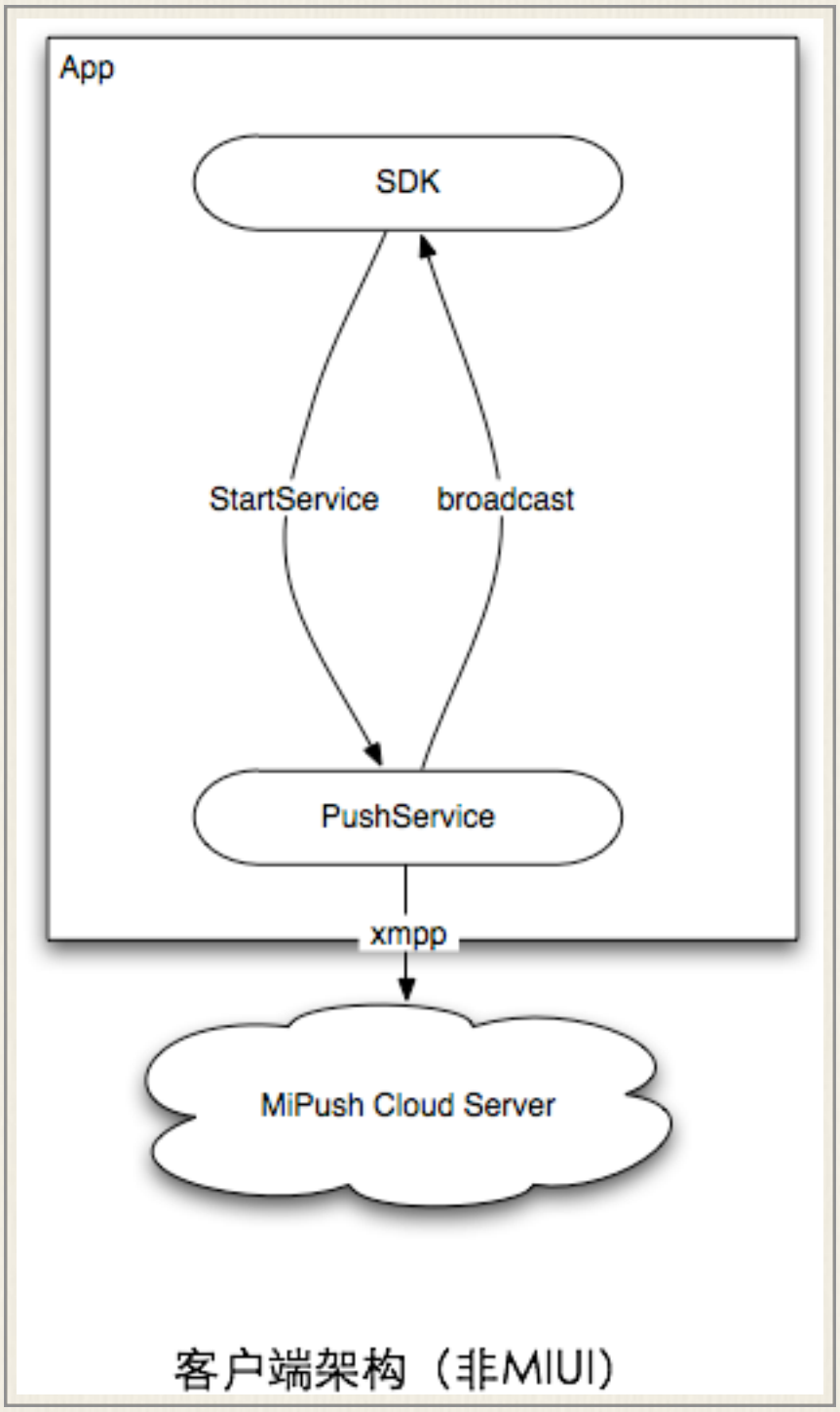
3. HTTP前端：这一层负责对接来自第三方App的服务器的发消息的HTTPS请求，以及来自客户端生成账号的HTTPS请求。

再就是数据存储，这里采用了小米的统一HBase存储，同时还使用MySQL来保存一些量不大，但需要复杂过滤条件的数据（topic等），并且为了降低对HBase的压力，中间还加了一层Redis作为缓存。

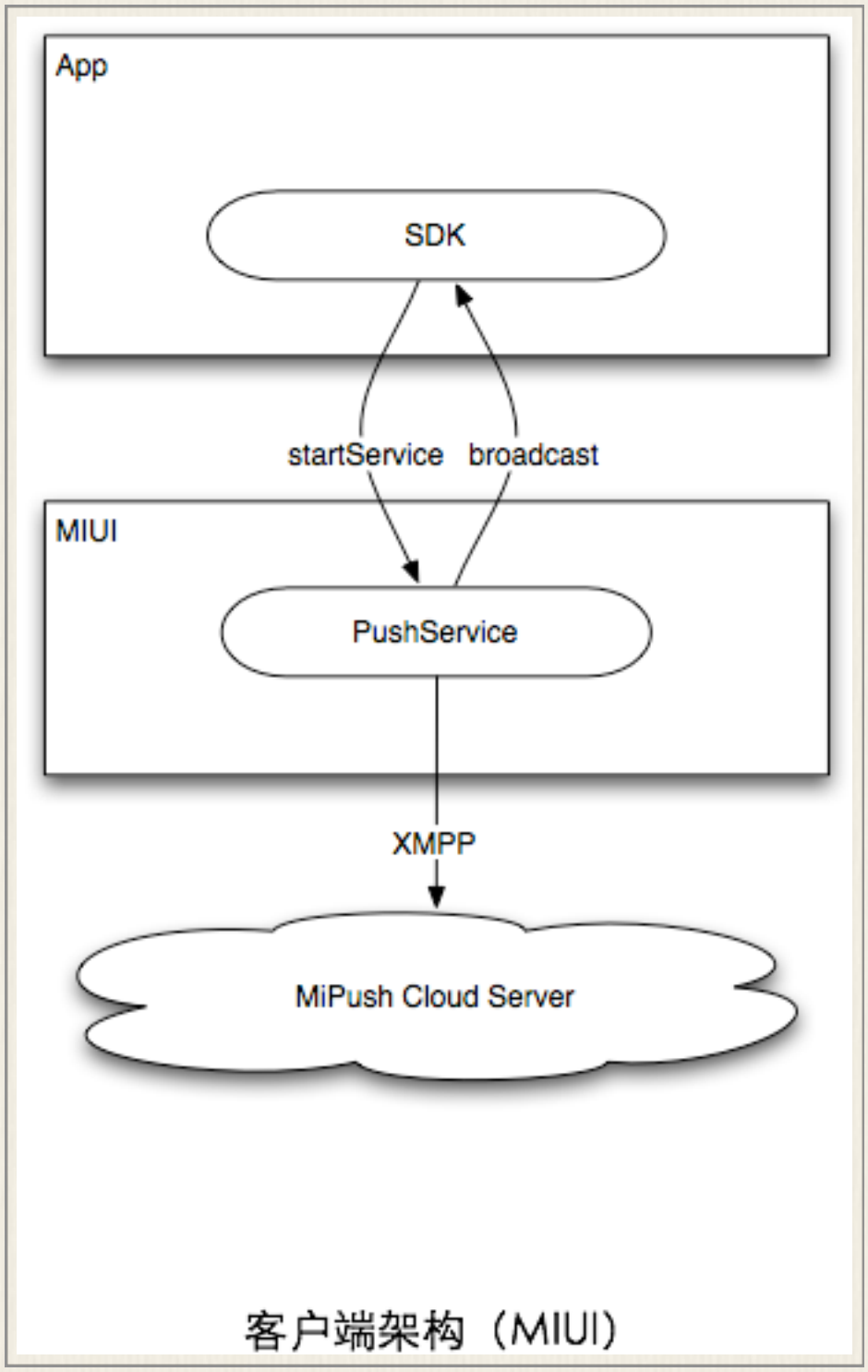


MiPush_Android整体架构图

最后看一下客户端架构。客户端SDK主要包含两个层次：SDK层和PushService层。前者提供了面向App接 入的接口、回调方法以及对Thrift的数据进行反序列化的处理逻辑；后者用于维护XMPP长连接和收发消息。两层之间使用Intent方式来传输数据。 值得一提的是，在MIUI系统上，PushService层是系统共用的，即MIUI系统提供了一个统一的PushService管理模块，不需要每个应 用单独启动自己的PushService。



客户端架构（非MIUI）



客户端架构（MIUI）

功能实现

小米推送服务支持单发和群发消息两种推送方式。单发消息支持针对regID和别名两 种方式，regID是小米推送服务后台根据设备标识+appID+时间戳生成，为了减少设备碰撞概率，设备标识我们采用的依据是imei+AndroidID+build序列号。别名是App在客户端设置上报的，便于应用将自己的设备/用户标识符同我们的regID作关联，这样 App就不需要在后台维护regID跟设备/用户的对应关系了。群发消息采用打标签的方式来区分，客户端和服务端都可以给指定设备设置标签，发消息的时 候，只需选取指

定标签发送即可，小米推送后台会将标签所对应的设备展开。一个标签支持的设备数无上限。

那小米推送服务的稳定性是如何保证的呢？小米推送服务采用多机房方案，平时流量均摊，一旦某个机房出现故障，流量无缝切换到其它机房，并且单个机房的容量能保证提供无损服务。目前是双机房部署，预计明年会扩展第三个机房。

安全性也是小米推送服务重点考虑的一个因素。数据传输过程中，得益于推送服务采用的 双层协议方案，消息会采取双重加密，第一重是XMPP传输层，保证数据在网络传输的过程中不会被篡改、监听。第二重是在Thrift二进制层，用以保证消 息到达Service之后，通过broadcast发送给App进程的过程中不会被截获和伪造。第二重加密往往会被其它第三方推送服务忽略，但其风险同样 很大。

性能指标

11.11大促，所面对的请求量是在小米推送服务的设计容量之内的，目前设计和机器规模可以支持峰值每分钟1000万条消息；平时业务量至少每分钟40万，峰值每分钟600万条消息。

推送消息量平时波动很大，所以开发团队准备着流量随时可能忽增200%的情况，并在线下做好压力测试和优化；如果流量特别大，还有以下应对措施：

1. 异步排队处理，此时消息送达时间可能会比平时稍慢，但不会对整个系统有太大冲击；
2. 消息有优先级，广播消息会以低优先级处理；
3. 限流，控制开发者发送消息的频率；
4. 扩容，如果机器负载过高或者某个服务有瓶颈，可以很快速地增加机器，部署服务，增强系统处理能力。

小米推送服务所经历的重构

软件系统在开发和演进过程中，经常会经历较大规模的重构。小米推送服务有两次比较大的重构。

一是开发语言从Erlang 转为Java。 小米原来的消息系统是使用Erlang开发的，所以推送系统的第一版也是基于Erlang；但是Erlang的社区不够活跃，开发人员很 难找，学习曲线陡，支持工具和类库少，所以后来开发团队选择了使用Java重新开发；迁移到Java后，对开发人员的要求降低，各种工具和类库较多，大大 提高了开发效率。

二是无处不在的Cache。客户端使用小米推送服务的SDK，开发者使用API的情况千变万化，很多场景是意料之外的； 需要对调用频繁的业务添加Cache，尽可能在本地进程内处理；例如，对于客户端调用API设置别名和订阅topic，先检查Cache是否已经设置过， 只有没有设置才往后端服务发送；优化后，后台服务的业务压力大大减少。

在开发小米推送过程中的一些感悟

1. 服务要支持水平扩展，尽可能实现为无状态，或者使用一致性哈希进行划分；方便扩容，可以保证即使系统暂时有性能瓶颈也能通过加机器解决。

2. 监控先行，能够很方便地采集、分析服务器的负载和业务的请求量、percentile、slow log，能够清楚了解到系统的瓶颈，有针对性地改进。

3. 不要过早优化，先实现功能并尽快上线，根据监控数据对关键地方进行优化。

4. 敏捷开发，快速迭代，日拱一卒，每天都有简短的站立会议，能够迅速响应变化，持续改进系统。

原文链接：http://www.infoq.com/cn/news/2014/11/xiaomi-1111-pushservice

内核层网络栈优化项目Fastsocket背后的故事

作者：刘宇 杨赛

2014年10月18日，当时就职于新浪操作系统团队的林晓峰在Github上开源了名为Fastsocket的项目，并在之后一天的中国Linux内核开发者大会上对该项目的原理和应用效果进行了介绍。根据Github官网的介绍，Fastsocket是：

- 高度可扩展的socket
- 是Linux内核层面的底层网络实现
- 在多核机器上可实现极佳性能，24核以内的性能增长呈线性，远超过默认内核在12核以上的机器就会出现性能下降的情况
- 非常容易使用和维护，应用代码无需变更
- 针对kernel-2.6.32-431.17.1.el6/CentOS-6.5的实现
- 已经在新浪的生产环境部署
- 由新浪的操作系统团队发起
- 清华大学操作系统实验室、Intel、哲思自由软件社区（Zeuux）对该项目均有支持
- 开源协议为GPLv2

开源之后的两周之内，该项目迅速收获了1800多个star和200多个fork，可以说成为了开源社区又一新的热点项目。近日，InfoQ编辑对Fastsocket的主要维护人员林晓峰、新浪操作系统团队的负责人李晓栋进行了邮件采访，了解有关Fastsocket项目的更多背景。

InfoQ：简单介绍一下**fastsocket**的开发背景吧。这个项目主要是你在新浪这边跟清华大学的操作系统实验室一起合作。一开始是在什么时候发起的？发起的动机是什么？与清华大学的操作系统实验室、**Intel**和哲思自由软件社区的合作模式是怎样的?

李晓栋：要说明清楚这点，需要从我们在新浪内部发起的FastOS计划谈起。

从技术上讲，FastOS 计划要做的是对Linux内核“协议栈、文件系统、IO”等不同子系统进行定向性的优化，以满足高性能网站的实际需要。Fastsocket是FastOS计划中的第一个子项目，今后我们还会推出FastTCP、FastIO……

从管理理念上讲，FastOS期望打造的是一个有公司保障、但没有公司边界、开放式的生态系统，可参与该计划的不仅有新浪，而且还包括：各类硬件提 供商、高校实验室、国内外自由软件组织、IT媒体、互联网技术同行以及对我们计划感兴趣的任何开发者。所有正式加入我们FastOS合作计划的组织和个 人，不仅可从共享通道中获得各合作方提供的软硬件及宣传资源，更为重要的是： 在对其它合作方利益无损的前提下，可以各取所需，实现合作成果的最大化分享。前面提到FastOS计划是有公司保障的，一是指该计划中所有的子项目都是在 新浪生产环境中被正式使用的，所有公开的测试数据都是真实的，有可信度方面的保障；二是指FastOS计划的日常运作管理由新浪操作系统团队做长期保障， 有一套明确的治理细节和管理流程，同时我们会充分考虑并切实避免合作方之间的利益冲突。 在这里，我们也诚邀请大家加入进来。

林晓峰：Fastsocket项目与2013年初正式立项，该项目最初要解决的问题就是要提升七层交换服务的Haproxy的单机性能。提升单机性 能根本原因在于降低成本，包括硬件相关成本和集群运维成本。经过Haproxy系统详尽分析后，我们发现大部分CPU资源消耗在kernel里，并且在多 核平台下，kernel在网络协议栈处理过程中存在着大量同步开销。据此，我们将开发kernel并行网络协议栈作为核心目标，来满足未来多核平台下万兆 高性能的网络需求。随着Fastsocket展现出强劲的性能优势，以及在新浪生产环境落地，我们希望可以将项目成果整理过学术论文，并有信心冲击国际顶 级系

统和网络方面的技术会议。借助合作伙伴Intel的牵线，我们联系到了清华操作系统中心的陈渝教授，我们一拍即合的开始Fastsocket项目的深 入合作，并且完成了Fastsock-
et论文，并已经向相关会议投稿，目前在等待结果。

InfoQ：如你之前的分享所说，多核机器在没有优化之前，CPU资源大多消耗在锁上面了。多线程的性能提升一般有哪些手段，各自的原理是什么?

林晓峰：我并不是多线程编程专家，不过可以给一些有关多核平台性能优化的通用建议。设计程序框架的时候，要尽可能的避免多线程访问需要同步的共享资 源，互斥上锁是多核平台性能第一杀手，每个线程只访问自己的数据是多核平台最高效，用户程序和系统内核里都一样。另外，线程数量不宜太多，并最好和核心绑 定，线程调度也是有开销的，保持线程在一个核心上运行可以让CPU cache更高效。

InfoQ：简单介绍一下Fastsocket提升性能的技术原理？做了哪些技术上的实现或优化?

林晓峰：Fastsocket提升性能，主要在于提高了kernel网络协议栈的效率，所以网络I/O密集的应用可以收到很好的性能提升效果。Fastsocket对网络协议栈内部优化，在github上的主线有总计7个优化特性。这些优化可以分为两个维度。

第一个维度是多核扩展性的优化，也就是让kernel网络协议栈在多核平台发挥多核的并行处理优势。这个维度又可以分为两个方向：一是，将网络协议 栈处理的关键数据结构做CPU核心间的隔离，使得每个核心有完全本地的访问数据，从而消除了执行路径上核心间的同步开销。二是，使得任意的某个TCP连接 的全部处理，都在一个核心上完成，这样可以最大化的提高CPU cache利用率。

第二个维度是单核性能的提升，也就是不考虑多核同步的情况下，如何提升网络协议栈在单个核心上的绝对性能。这个维度也可以分为两个方向：一是，将 kernel中的通用服务为网络I/O做专用定制，来提升网络协议栈的性能。二是，做网络协议栈的跨层优化，改变传统协议栈TCP/IP协议栈的严格分层 处理，将传输的关键信息垂直贯穿网络协议栈，来做全局的优化。

InfoQ：Nginx在CPU均衡上已很不错了，factsocket对于吞吐量的增加是如何实现的？后续是否考虑做成nginx modules？针对HAProxy的实现与Nginx一样吗？有什么差异？

林晓峰：Fastsocket是内核层面的优化，对用户程序是透明的，并不需要开发Nginx模块来支持。Fastsocket对应用程序来说通用的，提升的是内核网络协议栈的效率。

InfoQ：吞吐量的提升通用于4、7层代理吗？

林晓峰：Fastsocket对于网络性能的提升，适用于使用socket API来进行网络I/O的应用程序，所以我们将它命名为Fastsocket。如果4层代理是指LVS，那是Fastsocket是不适用的，因为LVS 功能是借助kernel里Netfilter框架实现的。

InfoQ：像Fastsocket这类以性能优化为主打的开源项目，品质保障、技术方案选择、成本管控非常重要，你能介绍一下这方面的经验吗？

李晓栋：在做性能优化的过程中，很多时候我们往往会陷入到仅关注性能指标的误区，而忽视了程序交付给运维人员后的部署成本、运维成本。比方说：是否 跟现有的上层软件、运维手段兼容？有无自动化的部署脚本？应急回滚是否方便？是否提供了良好的统计和追踪机制？等等等等。这些都需要在方案设计环节充分考 虑，否则很有可能出现“运维成本” 大于“性能提升所节省的硬件成本”，最终导致无法在生产环境中很好地使用起来。也就是说，我们需要在性能和可运维之间找到一个最佳平衡点。其实在Fastsocket之前我们还有一版内部代号为“Hydra”的预览版，性能比现在的Fastsocket要更好，但需要修改haproxy、 nginx等上层软件的代码，运维成本过高，因此被我们果断放弃了。关于品质保障，我觉得最好的办法就是，在设计测试用例时，要把生产环境中可能出现的各 类正常、非正常操作和情景尽量都考虑进去，最好能让比较资深的运维人员参与到测试用例设计中。当然，测试过程中，细心必不可少，要能敏锐捕捉异常情况。

InfoQ：Fastsocket开源之后立刻在Github上得到了上千个star，看来很多人也有这方面的需求。目前主要得到的反馈有哪些？

林晓峰：Fastsocket在Github上正式开源到现在刚两周多，到写稿时已经接近两千star，这是出乎我们意料的。

我们收到的反馈主要分为两方面。一是，具体是如何实现的，对其性能的大幅性能的技术点很有兴趣。二是，比较关注Fastsocket是否有移植到3.X kernel版本和合并到kernel主线的计划。在Haproxy的mailing list上也看到关于Fastsocket的讨论，很高兴的看到Haproxy作者对我们项目也很感兴趣，并表示可以考虑对Fastsocket进行直接 支持。

InfoQ：有其他公司的人来参与支持这个吗？未来以开源模式更新，对于项目的commit、review机制有什么计划?

林晓峰：据我个人所知，已经几家大型互联网公司对Fastsocket有试用的兴趣。目前项目的commit主要采用Github的pull request机制，由我来review代码。未来希望可以吸纳更多活跃的开发者作为committer，用社区方式去维护Fastsocket项目。

InfoQ：Fastsocket后续对新版本的内核、新版本的CentOS的支持，计划用怎样的方式去长期维持?

李晓栋：Fastsocket对不同版本内核和CentOS发行版的持续支持，采用两种方式：一是由新浪根据生产环境需要和操作系统使用策略，适时 升级，这种方式相对比较稳健和持续，但更新周期相对要长一些。第二种方式是依托社区的支持，目前已有热心贡献者愿意帮助我们将fastsocket移植到CentOS7下，在这里我们也表示深深感谢。

受访者简介

林晓峰，前新浪网高级系统开发工程师，关注网络，关注高性能，关注Linux内核。

李晓栋，新浪网研发中心高级技术经理，有十年的互联网工作经验，是“新浪软件负载均衡系统” 和“ 新浪操作系统管理与优化”方面的重要开拓者，也是FastOS计划和管理理念的提出者。

原文链接：http://www.infoq.com/cn/news/2014/11/fastsocket-github-opensource